VAX Pascal

digital

User Manual

User Manual

Order Number: AA-H485F-TE

VAX Pascal User Manual

Order Number: AA–H485F–TE

December 1989

This document contains information about selected programming tasks using the VAX Pascal programming language.

Revision/Update Information:	This revised document supersedes the information in *VAX Pascal User Manual* (Order No. AI–H485E–TE).
Operating System and Version:	VMS Version 5.2 or higher
Software Version:	VAX Pascal Version 4.0

digital equipment corporation
maynard, massachusetts

First Printing, November 1979
Revised, March 1985
Revised, February 1987
Revised, December 1989

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

© Digital Equipment Corporation 1979, 1985, 1987, 1989.

All Rights Reserved.

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA	MASSBUS	VAX RMS
DDIF	PrintServer 40	VAXstation
DEC	Q-bus	VMS
DECnet	ReGIS	VT
DECUS	ULTRIX	XUI
DECwindows	UNIBUS	
DIGITAL	VAX	
LN03	VAXcluster	digital™

The following is a third-party trademark:

PostScript is a registered trademark of Adobe Systems, Inc.

ZK4568

Contents

Figures

Tables

Preface

This document contains information about selected programming tasks using the VAX Pascal programming language. It contains information on using some VAX Pascal language elements in combination, and it provides examples of how to improve programming efficiency.

Intended Audience

This manual is intended for experienced applications programmers with a basic understanding of the Pascal language. Some familiarity with your operating system is helpful. This is not a tutorial manual for new Pascal users.

Document Structure

This manual consists of the following chapters and glossary:

- Chapter 1 describes the use of data types.
- Chapter 2 describes the use of procedures, of functions, and of parameter lists.
- Chapter 3 describes the use of separately compiled modules.
- Chapter 4 describes programming techniques that improve the efficiency of compilation and execution.
- Chapter 5 describes programming in the VMS environment.
- Glossary provides a glossary of VAX Pascal terminology.

Associated Documents

The following documents may also be useful when programming in VAX Pascal:

- *VAX Pascal Reference Manual*—Provides information on the syntax and semantics of the VAX Pascal programming language.
- *VAX Pascal Reference Supplement for VMS Systems*—Provides information on the programming environment beyond the scope of the VAX Pascal language syntax and semantics. It contains reference information on VMS operating-system features, VAX architecture information, and environment-specific affects of VAX Pascal language features.
- *VAX Pascal Installation Guide*—Provides information on how to install VAX Pascal on your operating system.
- VMS operating system manuals—Provide full information about the system. The *VMS Master Index* briefly describes all VMS system documentation, defines the intended audience for each manual, and provides a synopsis of each manual's contents.

Conventions

Table 1 presents the conventions used in this manual.

Table 1: Conventions Used in This Manual

Convention	Meaning
. . .	A vertical ellipsis in a figure or example means that not all of the statements are shown.
PROGRAM WRITELN	Uppercase letters and special symbols in syntax descriptions indicate reserved words and predeclared identifiers. For example: `BEGIN END`

(continued on next page)

Table 1 (Cont.): Conventions Used in This Manual

Convention	Meaning
Temp : INTEGER; PRED(n)	Lowercase letters represent user-defined identifiers or elements that you must replace according to the description in the text.
$ PASCAL $_File:	The hardcopy version of this manual has interactive examples that show user input in red letters and system responses or prompts in black letters. The online version displays user input in a bold font.
module	A term that appears in bold is defined in the glossary in this manual.

{
Source File: EXAMPLE.PAS
}
	Many of the code examples in this manual have a suggested source-file name in the comments at the top of the program. VAX Pascal ships some (but not all) of the examples in this manual on the kit; these files are named according to the source-file name presented in this book. For information on the location of these files, refer to the release notes.

In this manual, complex examples have been divided into several lines to make them easy to read. VAX Pascal does not require that you format your programs in any particular way.

Chapter 1

Data Types

This chapter discusses the following information:

- Predefined and user-defined data types (Section 1.1)
- Nesting Arrays and Records (Section 1.2)
- String types (Section 1.3)
- Nonstatic types (Section 1.4)
- Examples (Section 1.5)

NOTE

The sections at the beginning of this chapter use code fragments from the examples contained in the section at the end of the chapter. Complete code examples are located in Section 1.5.

For More Information:

For reference information on all data types, see the *VAX Pascal Reference Manual*.

1.1 Predefined and User-Defined Data Types

In VAX Pascal, a complete, valid data type determines the range of values, set of valid operations, and maximum storage necessary for any data item of that data type. In some instances, a single VAX Pascal predeclared identifier is sufficient to specify a complete data type, as with the following data types:

- CHAR
- INTEGER
- UNSIGNED

- REAL
- BOOLEAN

A pointer data type differs from the types in the previous list. A pointer type is a complete data type in and of itself (specified by the special symbol (^)); however, the pointer type requires the specification of a base type, which declares the data type of the object to which the pointer points. Consider the following:

```
VAR
   Ptr : ^Integer;   {Pointer to an integer base type}
```

There are some data types, called user-defined data types, that are not complete, valid data types until you specify more information in your program. The following data types are user-defined:

- Enumerated types
- Subrange types
- Array types
- Record types
- Set types
- File types
- Schema types

With all of these data types, VAX Pascal requires more information than a reserved word. For instance, the reserved word ARRAY does not provide enough information to form a complete data type. You must provide boundaries, field names, component types, or other information to form a complete type. Consider the following:

```
TYPE
   Array_Type = ARRAY[1..10] OF INTEGER;
VAR
   x : ARRAY;  {Invalid; what are the bounds and the element type?}
   y : ARRAY[1..10] OF INTEGER;  {One valid data type}
   z : ARRAY[1..10] OF INTEGER;  {Another valid data type}
   a : Array_Type;               {A third valid data type}
   b : Array_Type;               {Same type as that of a}
```

For More Information:

- On schema types (Section 1.4)
- On string types (Section 1.3)
- On arrays and records (Section 1.2)

1.2 Nesting Arrays and Records

If your application requires, you can nest arrays within arrays and records within records; arrays can have ARRAY element types and records can have RECORD field types. Consider the following:

```
TYPE
   My_Array = ARRAY[1..10] OF CHAR;
   TwoD_Array = ARRAY[1..3] OF My_Array;   {Two-dimensional array}

   {Other ways to declare two-dimensional arrays:}
   TwoD_Array2 = ARRAY[1..3, 1..10] OF CHAR;
   TwoD_Array3 = ARRAY[1..3] OF ARRAY[1..10] OF CHAR;
   ThreeD_Array = ARRAY[1..5] OF TwoD_Array; {Three-dimensional array}

   Personal_Record = RECORD
      Name : STRING( 30 );
      Age  : INTEGER;
      END;
   School_Record = RECORD
      Year : ( FR, SO, JR, SR );
      Suspended : BOOLEAN;
      Student   : Personal_Record;    {Nested Record}
      END;
```

Example 1–1 uses a two-dimensional array to create a program that allows two users to play tic tac toe. For instance, the following code fragment from the larger example shows the declaration and initialization of the tic tac toe board:

```
TYPE
   Matrix  = ARRAY[1..3, 1..3] OF CHAR;
VAR
   Board       : Matrix VALUE [ 1..3: [ OTHERWISE ' ' ] ];
{In the executable section:}
Board[ 1, 2 ] := 'x';          {Column 1, Row 2}
{Equivalently, you can use this syntax:}
Board[1][2] := 'x';
```

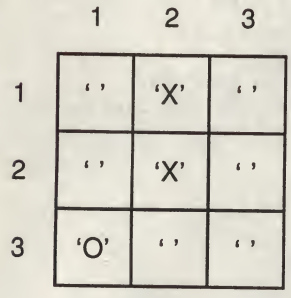
The array constructor in the previous code specifies the same array constructor for all three elements of the first dimension of the array.

Figure 1–1 illustrates the tic tac toe board. When running the program, the following entries would create the matrix in Figure 1–1:

```
Welcome to the Tic Tac Toe Game.
Enter the number 0 to QUIT.

Player 1 (X), please enter column and row # :  1 2
Player 2 (O), please enter column and row # :   3 1
```

```
Player 1 (X), please enter column and row # :   2 2
   .
   .
   .
```

Figure 1–1: Multidimensional Array as a Tic Tac Toe Board



ZK–1474A–GE

Sometimes the field specifications for nested records can get quite long. Consider the following from Example 1–2:

```
{The corresponding type declarations:}
TYPE
   Grade_List( Num_Grades : INTEGER ) =
      ARRAY [1..Num_Grades] OF INTEGER;

   Student = RECORD
      Name        : PACKED ARRAY [1..30] OF CHAR;
      Test_Scores : Grade_List( Get_Num_Grades ); {Discriminate}
      END;

   Class_Grades( Num_Students : INTEGER ) =
      ARRAY [1..Num_Students] OF Student;

VAR
   Current_Class : Class_Grades( Get_Num_Students );  {Discriminate}

   {In the executable section:}
Final_Grade := Total / Current_Class[i].Test_Scores.Num_Grades;
```

If your application would benefit, you can use the WITH statement to make it easier to read field specifications from nested records or to read actual discriminant values from variables of schema types. For instance, the

following WITH statement reduces the code necessary for the compiler to determine the specified field and actual discriminant:

```
WITH Current_Class[i], Test_Scores DO
   Final_Grade := Total / Num_Grades;
```

The variable Current_Class[i] is of a RECORD type and the variable Test_Scores is a discriminated schema type of a field within Current_Class[i]. VAX Pascal is able to determine the context of Test_Scores from Current_Class[i] and is able to determine the context of Num_Grades from Test_Scores.

1.3 String Types

When comparing strings, you can use either the non-blank padding string functions or the VAX Pascal relational operators. The functions do not blank-pad strings before comparing, (the effect of using these functions is similar to the effect of using Ada relational operators) and VAX Pascal does blank pad when you use the relational operators. Consider the following:

```
EQ('hello', 'hello  ')       {Returns FALSE}
('hello' = 'hello  ')        {Evaluates to TRUE}

GE('hello', 'hello  ')       {Returns FALSE}
('hello' >=  'hello  ')      {Evaluates to TRUE}

GT('hello  ', 'hello')       {Returns TRUE}
('hello  ' > 'hello')        {Evaluates to FALSE}

LE('hello  ', 'hello')       {Returns FALSE}
('hello  ' <= 'hello')       {Evaluates to TRUE}

LT('hello', 'hello  ')       {Returns TRUE}
('hello' < 'hello  ')        {Evaluates to FALSE}

NE('hello', 'hello  ')       {Returns TRUE}
('hello' <> 'hello  ')       {Evaluates to FALSE}
```

If you are comparing the equality or inequality of very large strings, it is sometimes more efficient to use the EQ and NE functions instead of the = and <> operators. When using the operators, VAX Pascal compares each string, character by character, until VAX Pascal detects either a difference or the end of one string. When you use the functions, VAX Pascal sometimes detects different string lengths without comparing the strings character by character.

When trying to determine the length of a string, you can use either the LENGTH function, or the .LENGTH component of a STRING or VARYING

OF CHAR type. The LENGTH routine and .LENGTH provide the same value. Consider the following:

```
VAR
   One_String : STRING( 25 ) VALUE 'Harvey Fierstein';
{In the executable section:}
WRITELN( One_String.LENGTH, LENGTH( One_String ) );
```

The WRITELN call writes 16 and 16 to the predeclared file OUTPUT (by default, to your terminal).

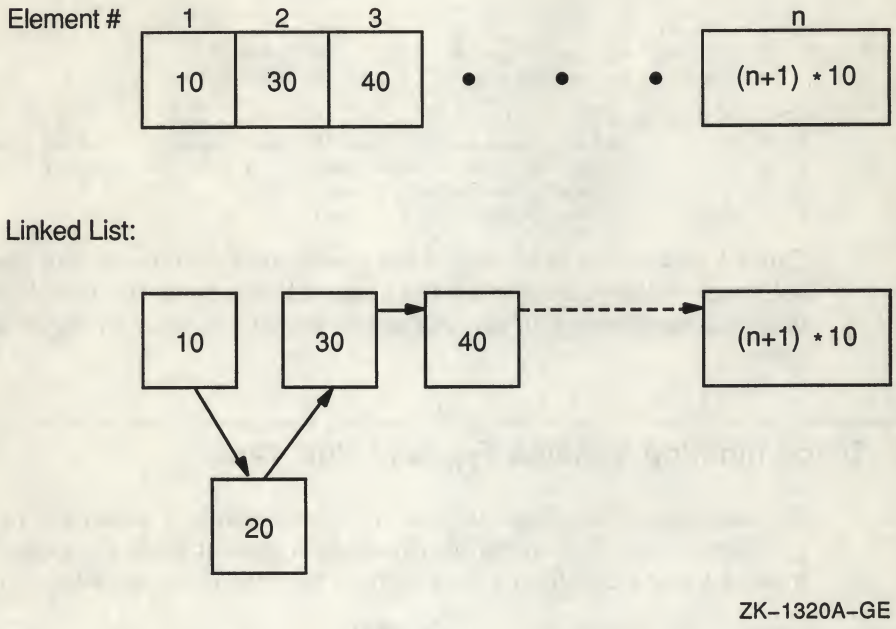
1.4 Nonstatic types

Nonstatic types are types that contain a schema type in its type declaration; in many instances, these types are not complete, valid data types until run time, since actual discriminants can be run-time values. For instance, the following types are nonstatic:

```
TYPE
   Array_Template( Upper : INTEGER ) =
      ARRAY[1..upper] OF INTEGER;
   Array_Type1 = Array_Template( 10 );
   Array_Type2 = Array_Template( Func_Call );

   Sub_Range( a, b : INTEGER ) = a..b;
   My_Array = ARRAY[ Sub_Range( 1, 10 ), Sub_Range( 1, 20 )] OF CHAR;

   My_Record = RECORD
      f1 : INTEGER;
      f2 : STRING( 4 );
      END;

   My_Record = RECORD
      f1 : INTEGER;
      f2 : Array_Type1;
      END;

   My_Set = SET OF Sub_Range( 1, 25 );
```

The addition of schema types to the language may affect implementation decisions, such as the decision when choosing between arrays and linked lists. Using previous versions of VAX Pascal, you needed to create a linked list to contain units of data when the total number of units was known only at run time. Schema types allow you to do this. If you need to add or to delete units in the middle of the list, then linked lists would still be the algorithm of choice; if you only need to add units to the end of the list, a schema type array might be easier to understand and to use. Figure 1–2 illustrates the difference between the two constructs.

Figure 1–2: Schema Array Types and Linked Lists



For More Information:

For information on schema parameters, see Section 2.1.1.

1.4.1 Schema Families and Type Compatibility

If two data types are derived from the same schema type, then the two created data types are considered to be of the same **schema family** as the original schema type. Consider the following:

```
TYPE
   My_Subrange( a, b : INTEGER ) = a..b;
   Sub_A = My_Subrange( 1, 5 );
   Sub_B = My_Subrange( 1, 5 );
   Sub_C = My_Subrange( -50, 50 );
```

The types Sub_A, Sub_B, and Sub_C are all of the My_Subrange schema-type family. Consider the following:

```
TYPE
   My_Subrange( a, b : INTEGER ) = a..b;
   My_Array( Upper : INTEGER ) = ARRAY[1..Upper] OF INTEGER;
VAR
   i : My_Array( 10 );
   j : My_Array( 10 );
   k : My_Array( 15 );
   l : ARRAY[ My_Subrange( 1, 10 ) ] OF INTEGER;
   m : ARRAY[ My_Subrange( 1, 10 ) ] OF INTEGER;

{In the executable section:}
i := j;       {Legal; same schema family, same actual discriminant}
i := k;       {Illegal; same schema family, different actual}
i := l;       {Illegal; different types}
l := m;       {Illegal; different types}
```

Types l and m are not assignment compatible despite having the same subrange values specified by the same schema type; the two distinct type declarations create two distinct types, regardless of the ranges of the two types.

1.4.2 Discriminating Schema Types at Run Time

If your application requires, you can discriminate a schema type with a run-time value. For instance, this code fragment from Example 1–2 uses integer values in a file to discriminate schema types, as follows:

```
FUNCTION Get_Num_Grades   : INTEGER;   {Function body}
FUNCTION Get_Num_Students : INTEGER;   {Function body}

TYPE
   Student = RECORD
      Name        : PACKED ARRAY [1..30] OF CHAR;
      Test_Scores : Grade_List( Get_Num_Grades ); {Discriminate}
      END;
   Class_Grades( Num_Students : INTEGER ) =
      ARRAY [1..Num_Students] OF Student;
VAR
   Current_Class : Class_Grades( Get_Num_Students );  {Discriminate}
```

If you need to declare global variables, constants, or types for use in the functions, the Extended Pascal standard allows you to place additional CONST, VAR, or TYPE sections before the function declarations. You can then place all remaining variables, constants and types in sections closer to the executable section of your program.

If you choose, you can use a variable to discriminate a schema type. Consider the following:

```
VAR
   a : [EXTERNAL] INTEGER;
TYPE
   My_Array( Upper ) = ARRAY[1..Upper] OF INTEGER;
   One_Array_Type = My_Array( a );   {Determined after compile time}
```

1.5 Data-Type Examples

Example 1–1 shows a tic tac toe game that uses a multidimensional array.

Example 1–1: Tic Tac Toe Using a Multidimensional Array

```
{
Source File: MULTI_DIMEN_ARRAY.PAS
This program uses a multidimensional array to play a game of tic tac toe.
}
PROGRAM Tic_Tac_Toe( INPUT, OUTPUT );
CONST
   Player_1 = 'x';
   Player_2 = 'o';
   Blank    = ' ';
TYPE
   Matrix  = ARRAY[1..3, 1..3] OF CHAR;
VAR
   Column, Row : INTEGER;
   Quit        : BOOLEAN VALUE FALSE;
   Marker      : ARRAY[1..2] OF CHAR VALUE [1: 'x'; 2: 'o'];
   Next_Player : ARRAY[1..2] OF 1..2 VALUE [1: 2; 2: 1];
   Player      : 1..2 VALUE 1;
   {
The following constructor specifies 3 elements, each of which is an
array of blanks.
   }
   Board       : Matrix VALUE [ 1..3: [ OTHERWISE ' ' ] ];
```

(continued on next page)

Example 1–1 (Cont.): Tic Tac Toe Using a Multidimensional Array

```
{Check to see if someone has won:}
FUNCTION Check_For_Win( VAR Board : Matrix ) : BOOLEAN;
   VAR
      Winner, Space : CHAR;
   BEGIN
   Winner := Blank;  {Until 'x' or 'o' wins}
   IF Board[1,1] <> Blank THEN {If 'x' or 'o', check for win}
      BEGIN
      Space := Board[1,1];
      IF ( ( Space = Board[1,2] ) AND ( Space = Board[1,3] ) ) OR
         ( ( Space = Board[2,2] ) AND ( Space = Board[3,3] ) ) OR
         ( ( Space = Board[2,1] ) AND ( Space = Board[3,1] ) ) THEN
         Winner := Space;
      END;  {IF Board[1,1]}

   IF ( Board[2,1] <> Blank ) AND ( Winner = Blank) THEN
      BEGIN
      Space := Board[2,1];
      IF ( ( Space = Board[2,2] ) AND ( Space = Board[2,3] ) ) THEN
         Winner := Space;
      END;  {IF Board[1,2]}

   IF ( Board[3,1] <> Blank ) AND ( Winner = Blank) THEN
      BEGIN
      Space := Board[3,1];
      IF ( ( Space = Board[2,2] ) AND ( Space = Board[1,3] ) ) OR
         ( ( Space = Board[3,2] ) AND ( Space = Board[3,3] ) ) THEN
         Winner := Space;
      END; {IF Board[3,1]}

   IF ( Board[3,2] <> Blank ) AND ( Winner = Blank) THEN
      BEGIN
      Space := Board[3,2];
      IF ( ( Space = Board[2,2] ) AND ( Space = Board[1,2] ) ) THEN
         Winner := Space;
      END; {IF Board[3,2]}

   IF ( Board[3,3] <> Blank ) AND ( Winner = Blank) THEN
      BEGIN
      Space := Board[3,3];
      IF ( ( Space = Board[2,3] ) AND ( Space = Board[1,3] ) ) THEN
         Winner := Space;
      END; {IF Board[3,3]}
```

(continued on next page)

Example 1–1 (Cont.): Tic Tac Toe Using a Multidimensional Array

```
   CASE Winner OF
      Player_1    :  BEGIN
                     WRITELN;
                     WRITELN( 'Tic, Tac, Toe! Player 1 wins!' );
                     Check_For_Win := TRUE;
                     END;
      Player_2    :  BEGIN
                     WRITELN;
                     WRITELN( 'Tic, Tac, Toe! Player 2 wins!' );
                     Check_For_Win := TRUE;
                     END;
      Blank       :  BEGIN
                     Check_For_Win := FALSE;
                     END;
      END;  {CASE}
   END; {Of FUNCTION}

BEGIN  {Main program}
WRITELN;
WRITELN( 'Welcome to the Tic Tac Toe Game.' );
WRITELN( 'Enter the number 0 to QUIT.' );
WRITELN;

REPEAT
   WRITE( 'Player ', Player:1, ' (', Marker[Player],
          '), please enter column and row # :  ' );
   READ( Column );
   READ( Row );
   WRITELN;
   IF ( Column = 0 ) OR ( Row = 0 ) THEN
      Quit := TRUE
   ELSE
      BEGIN
      Board[ Column, Row ] := Marker[Player];
      Quit := Check_For_Win( Board );
      END;
   IF Player = 2 THEN WRITELN;
   Player := Next_Player[ Player ];
UNTIL Quit;

WRITELN;
WRITELN( 'Thank you for playing.' );
END.
```

Sample output from Example 1–1 is as follows:

```
Welcome to the Tic Tac Toe Game.
Enter the number 0 to QUIT.

Player 1 (X), please enter column and row # :  1 1
Player 2, please enter column and row # :   2 1

Player 1 (X), please enter column and row # :   1 2
Player 2, please enter column and row # :   3 1

Player 1 (X), please enter column and row # :   1 3

Tic, Tac, Toe! Player 1 wins!

Thank you for playing.
```

Example 1–2 shows two schema types that are discriminated at run time using function calls.

Example 1–2: Discriminating Schema Types with Function Calls

```
{
Source File: SCHEMA_FUNC.PAS
This example reads student's names and grades from a data file called
DATA.DAT.  This example uses schema types that are discriminated using
function calls; the function calls obtain the number of students and
number of grades to be read.
}
PROGRAM Process_Data( INPUT, OUTPUT, Input_File );
```

(continued on next page)

Example 1–2 (Cont.): Discriminating Schema Types with Function Calls

```
VAR
   Input_File      : TEXT;
   i, j, Total     : INTEGER;
   Final_Grade     : REAL;
   Temp_String     : STRING( 80 );
{
Open the file.  The first number in the file is "the number of grades
per student."
}
FUNCTION Get_Num_Grades: INTEGER;
   VAR
      Temp_Grades : INTEGER;
   BEGIN
   OPEN( Input_File, 'data.dat', HISTORY := OLD );  {Open the file}
   RESET( Input_File );
   READ( Input_File, Temp_Grades );      {Read the number of grades}
   Get_Num_Grades := Temp_Grades;
   END;
{
The second number in the file is "the number of students" in the class.
}
FUNCTION Get_Num_Students: INTEGER;
   VAR
      Temp_Students : INTEGER;
   BEGIN
   READLN( Input_File, Temp_Students );   {Get number of students}
   Get_Num_Students := Temp_Students;
   END;

TYPE
   {
Using schema types the number of grades and students (Num_Grades and
Num_Students) can be provided at run time by function calls.
   }
   Grade_List( Num_Grades : INTEGER ) =
      ARRAY [1..Num_Grades] OF INTEGER;

   Student = RECORD
      Name         : PACKED ARRAY [1..30] OF CHAR;
      Test_Scores : Grade_List( Get_Num_Grades ); {Discriminate}
      END;

   Class_Grades( Num_Students : INTEGER ) =
      ARRAY [1..Num_Students] OF Student;

VAR
   Current_Class : Class_Grades( Get_Num_Students );  {Discriminate}
```

(continued on next page)

Example 1–2 (Cont.): Discriminating Schema Types with Function Calls

```
BEGIN  {Main program}
{
Read data into variable of schema type using the value of the actual
discriminant Num_Students:
}
FOR i := 1 TO Current_Class.Num_Students DO
   BEGIN
   READ( Input_File, Current_Class[i].Name );
   {
Use value of actual discriminant Num_Grades to get grades:
   }
   FOR j := 1 TO Current_Class[i].Test_Scores.Num_Grades DO
      READ( Input_File, Current_Class[i].Test_Scores[j] );
   READLN( Input_File );
   END;  {FOR i}

CLOSE( Input_File );       {No more data needed}

{Calculate and write final grades:}
FOR i := 1 TO Current_Class.Num_Students DO
   BEGIN
   Total := 0;
   FOR j := 1 TO Current_Class[i].Test_Scores.Num_Grades DO
      Total := Total + Current_Class[i].Test_Scores[j];
   Final_Grade := Total / Current_Class[i].Test_Scores.Num_Grades;
   WRITELN( Current_Class[i].Name, ' has a final grade of ',
            Final_Grade:2:0 );
   END; {FOR i}
END. {PROGRAM Process_Data}
{
Sample data file DATA.DAT:

3 4
Rita Mae Brown                 100 90 89
Virginia Appuzzo               100 100 92
William Dannemeyer             50 62 59
John Briggs                    80 70 75
}
```

Sample output from Example 1–2 is as follows:

```
Rita Mae Brown                  has a final grade of 93.
Virginia Appuzzo                has a final grade of 97.
William Dannemeyer              has a final grade of 57.
John Briggs                     has a final grade of 75.
```

Example 1–3 shows how to build and to access a binary tree using VAX Pascal data types.

Example 1–3: Building and Accessing a Binary Tree

```
{
Source File: BTREE_NO_RECURSION.PAS
This program stores names and telephone numbers from the data file
DATA.DAT, and provides this information to users upon request.  This
code contains a schema type (Tree_Node).
}
PROGRAM Binary_Tree( INPUT, OUTPUT );

TYPE
   Ptr_Tree_Node = ^ Tree_Node;
   Tree_Node( Name_Length, Number_Length : INTEGER ) = RECORD
      Left_Link,
      Right_Link  : Ptr_Tree_Node VALUE NIL;
      Name        : PACKED ARRAY[1..Name_Length] OF CHAR;
      Number      : PACKED ARRAY[1..Number_Length] OF CHAR;
      END;
VAR
   Name         : VARYING [132] OF CHAR;
   Number       : VARYING [132] OF CHAR;
   In_File      : FILE OF VARYING [132] OF CHAR;
   Root         : Ptr_Tree_Node VALUE NIL;
   Node         : Ptr_Tree_Node;

PROCEDURE Add_Node( Name      : VARYING [Name_Bound] OF CHAR;
                    Number    : VARYING [Number_Bound] OF CHAR );
   VAR
      Done             : BOOLEAN;
      Current          : Ptr_Tree_Node;
      New_Node         : Ptr_Tree_Node;

   BEGIN {Non-recursive implementation }
   {Create node:}
                  {Storage is only as big as data requires:}
   NEW( New_Node, Name.LENGTH, Number.LENGTH );
   New_Node^.Name := Name;
   New_Node^.Number := Number;
```

(continued on next page)

Example 1–3 (Cont.): Building and Accessing a Binary Tree

```
   {Insert node in tree:}
   IF Root = NIL THEN
      Root := New_Node
   ELSE
      BEGIN
      Current := Root;
      Done := False;
      REPEAT
         IF Name = Current^.Name THEN  {If duplicate name, stop}
            BEGIN
            WRITELN( 'Duplicate name: ', Name );
            HALT;
            END
         ELSE  {Insert new node in alphabetical order, left to right}
            IF Name < Current^.Name THEN
               BEGIN
               IF Current^.Left_Link = NIL THEN
                  BEGIN
                  Current^.Left_Link := New_Node;
                  Done := True;
                  END
               ELSE Current := Current^.Left_Link;
               END  {IF Name < Current^.Name}
            ELSE
               BEGIN
               IF Current^.Right_Link = NIL THEN
                  BEGIN
                  Current^.Right_Link := New_Node;
                  Done := True;
                  END
               ELSE Current := Current^.Right_Link;
               END; {ELSE of IF Name < Current^.Name}
      UNTIL Done; {Search tree until insertion made or duplicate found}
      END;  {ELSE of IF Root = NIL}
   END; {of Add_Node}

FUNCTION Find_Node(
      Current : Ptr_Tree_Node;
      Name    : VARYING [Name_Bound] OF CHAR ) : Ptr_Tree_Node;

   VAR
      Done  : BOOLEAN;
```

(continued on next page)

Example 1–3 (Cont.): Building and Accessing a Binary Tree

```
   BEGIN
   Current := Root;
   Done := False;
   REPEAT
      IF Name = Current^.Name THEN  {If found, return node}
         BEGIN
         Find_Node := Current;
         Done := True;
         END
      ELSE  {Search tree left to right}
         IF Name < Current^.Name THEN
            IF Current^.Left_Link = NIL THEN
               BEGIN
               Find_Node := NIL;
               Done := True;
               END
            ELSE Current := Current^.Left_Link
         ELSE  IF Name > Current^.Name THEN
                  IF Current^.Right_Link = NIL THEN
                     BEGIN
                     Find_Node := NIL;
                     Done := True;
                     END
                  ELSE Current := Current^.Right_Link;
   UNTIL Done; {Search tree until NIL or found}
   END; {of Find_Node}

BEGIN {Program Binary_Tree}
{Read data file and create tree:}
OPEN( FILE_VARIABLE     := In_File,
      FILE_NAME         := 'data.dat',
      HISTORY           := OLD );
RESET( In_File );

WHILE NOT EOF( In_File ) DO
   BEGIN
   READ( In_File, Name );
   READ( In_File, Number );
   Add_Node( Name, Number )
   END;
CLOSE( In_File );
```

(continued on next page)

Example 1–3 (Cont.): Building and Accessing a Binary Tree

```
{Search for requested names:}
WRITE( 'Name: ' );
WHILE NOT EOF( Input ) DO
   BEGIN
   READLN( Name );
   Node := Find_Node( Root, Name );
   IF Node = NIL THEN
      WRITELN('Not Found')
   ELSE
      WRITELN( Node^.Number );
   WRITELN;
   WRITE('Name: ');
   END; {Of WHILE statement}
END.
{
Sample input from DATA.DAT:

Virginia Appuzzo
555-6767
Amy Ray
555-8997
Phillip Romanovsky
555-0182
}
```

Sample output from Example 1–3 is as follows:

```
Name:  Amy Ray
555-8997

Name:  Emily Saliers
Not Found

Name:  Virginia Appuzzo
555-6767

Name:  virginia appuzzo
Not Found

Name:  CTRL/Z
```

Chapter 2

Routines

VAX Pascal allows you to divide your program into subprograms called functions and procedures. Functions and procedures are collectively called routines. This chapter discusses the following topics about routines:

- Value and variable parameters (Section 2.1)
- Static and automatic variable allocation (Section 2.2)
- Structured function-return values (Section 2.3)
- Recursion (Section 2.4)
- Routine examples (Section 2.5)

NOTE

The sections at the beginning of this chapter use code fragments from the examples contained in the section at the end of the chapter. Complete code examples are located in Section 2.5.

For More Information:

For information on routines, on parameters, and on variable allocation, see the *VAX Pascal Reference Manual*.

2.1 Value and Variable Parameters

Parameters allow you to exchange data with a routine. You can conceive of actual parameters as being input for the routine. A formal parameter can be seen as a placeholder for the incoming data provided by the actual parameter.

When choosing between value and variable formal parameters, you need to decide whether you want the routine to do one of the following:

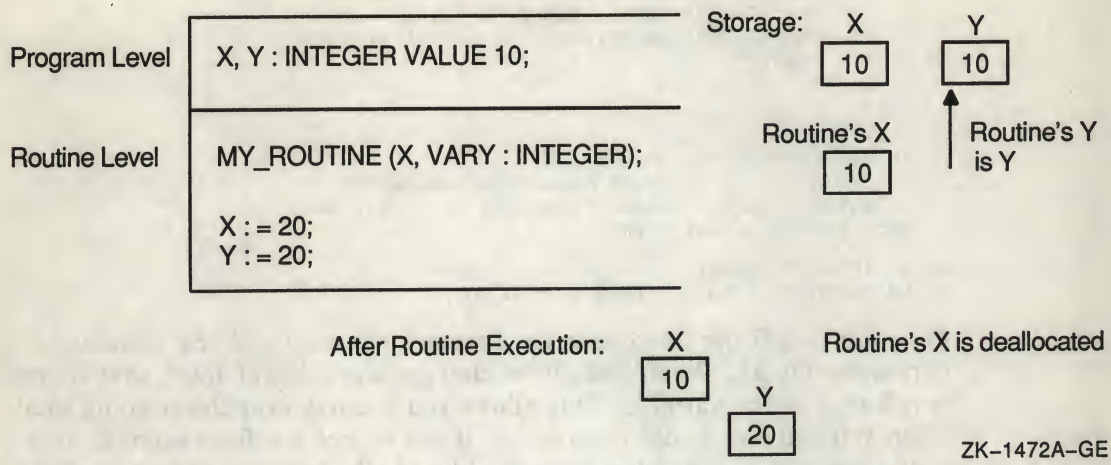
- Take the value of the actual parameter for internal use and leave its value unaltered after the routine call
- Alter the value of the actual parameter

If you want to maintain the original value of the actual parameter, use value parameters (you need no additional parameter syntax to specify this parameter type). In this instance, VAX Pascal passes the address of the actual parameter to the called routine. Upon execution of that routine, VAX Pascal copies the value to storage that is local to the routine, makes changes to the local variable, and deallocates the local storage at the end of routine execution. Throughout this process, the value of the actual parameter remains untouched by the called routine.

If you want the changes made within the routine to be reflected in the actual parameter's value, use variable parameters (by placing the reserved word VAR in front of the formal parameter). In this instance, VAX Pascal allows access to the storage of the actual parameter and allows the routine to treat the actual parameter as a variable. All changes made to the formal parameter within the routine are made to the value of the actual parameter. After routine execution, the value of the actual parameter reflects the changes made within the routine.

Figure 2–1 illustrates the use of these two parameters. Note that the figure assumes that you call My_Routine from the program level.

Figure 2–1: Value and Variable Parameters



In Figure 2–1, the assignment of 20 to y is reflected in the value of the actual parameter y after execution of My_Routine, since the corresponding formal parameter is a variable parameter. In the case of x, the assignment of 20 to x takes place to an object that is local to My_Routine; VAX Pascal deallocates storage for that object after execution of My_Routine.

Another way to conceptualize the differences in these types of parameters is to look at actual value parameters as being similar to read-only objects and to look at actual variable parameters as read/write objects.

Consider the following code fragment from Example 2–3:

```
PROGRAM Burger_Recursion( INPUT, OUTPUT );
VAR
   Total          : REAL VALUE 0.0;      {Customer's total}
   Done           : BOOLEAN VALUE TRUE; {Done with 1 order}

FUNCTION Get_Order( VAR Total : REAL;
                    Done      : BOOLEAN ) : INTEGER;
   VAR
      Not_Done_Yet  : BOOLEAN VALUE FALSE;     {Still ordering}
```

```
    BEGIN
    READLN( Item );
    CASE Item OF
       1  : BEGIN
            Total := Total + 0.75;
            Get_Order( Total, Not_Done_Yet );
            WRITELN( ' Hamburger        .75' );
            END;
       .
       .
       .
    IF Done AND ( Item <> 0 ) THEN
       BEGIN          {Print total for 1 order}
       Total := 0.0;  {Reset total for next customer}
    END; {Function Get_Item}

BEGIN {Main Program}
WHILE Get_Order( Total, Done ) <> 0 DO ;
```

Function Get_Order has one value parameter (Done) and one variable parameter (Total). When Get_Order changes the value of Total, that change is reflected in the variable. This allows you to carry over the running total when you call Get_Order recursively. If you do not set Total equal to zero at the end of a single order, Total would add all additional orders (just like the variable Get_Days_Total, which is in Example 2–3 but not shown in this code).

When you are designing your programs, remember that there are several different ways to reflect the work done in a routine throughout the rest of the compilation unit or application. The method that you choose depends on the requirements of the application. For instance, you can use any of these methods to reflect work done in a routine throughout the rest of the program:

- Accessing variables in an enclosing scope. Consider the following:

```
PROGRAM Example;
VAR
   x : INTEGER VALUE 15;

PROCEDURE Test;
   BEGIN
   x := 25;
   END;
```

  Some program designs discourage this technique because it can be confusing to determine where the variable is declared. This may cause problems for future maintenance. However, for some programmers and for some applications, uplevel variables can provide efficient solutions. After a call to Test in the previous code, x has the value 25.

- Using a function return value. Consider the following:

```
FUNCTION Age : INTEGER;
   BEGIN
   Age := 25;
   END;

{In the program executable section:}
Current_Age := Age;
```

  If reflecting a change to a single value is important to your application, you may wish to use a function instead of a variable parameter. Functions allow only one return value, but using them can provide a clarity that makes the code easier to maintain. Also, you can specify variable parameters to functions, if you need more values to be reflected after function execution. The design of your application would determine which values are good candidates for function return values and which are good candidates for variable parameters.

- Using external variable references. Consider the following:

```
PROCEDURE Test;
   VAR
      x : [EXTERNAL] INTEGER;
   BEGIN
   x := 25;
   END;
```

  This method allows you to reflect changes to the variable in other compilation units. (If you require code that you must port to another vender's compiler, you cannot use attributes.)

For More Information:

- On using foreign actual parameters (Section 5.3)
- On automatic and static variable allocation (Section 2.2)
- On compilation units (Chapter 3)
- On examples of routine parameters (*VAX Pascal Reference Manual*)

2.1.1 Parameters of Schema Types

To pass an actual variable parameter of a schema type to a formal parameter of a schema type, the two parameters have to be of the same schema family and must have the same actual discriminant values; if the formal parameter

is not discriminated, the actual parameter can have any discriminant value. For instance, consider the following code fragment from Example 2–1:

```
TYPE
   Student_List_Template( a, b : INTEGER ) =
      ARRAY[a..b] OF Student_Record;
   Student_List = Student_List_Template( 1, 5 );
VAR
   Students : Student_List;

PROCEDURE Get_Students( VAR Students : Student_List_Template );
   {Procedure body...}
PROCEDURE Write_Students( Students : Student_List );
   {Procedure body...}
```

Any actual parameter passed to Get_Students must be of the Student_List_Template family but can have any discriminant value. An actual parameter passed to Write_Students must be of the Student_List_Template family and must have actual discriminants values of 1 and 5.

When you pass a varying-length string expression to an undiscriminated STRING parameter of value semantics, the current length of the actual parameter (not its declared maximum length) becomes both the current length and the maximum length of the formal parameter. When you pass a discriminated STRING variable to a VAR parameter, the declared maximum length of the actual parameter becomes the maximum length of the formal parameter.

When specifying formal STRING parameters in one parameter section (separated by commas), the current size of the actual string parameter determines compatibility as opposed to the value of the actual discriminants. Consider the following:

```
VAR
   One_String, Two_String : STRING( 20 );

PROCEDURE Test_Strings( One, Two : STRING ); {Procedure Body...}

{In the executable section:}
One_String := 'Hello Blanche';
Two_String := 'But ya ahh, Blanche!';
Test_Strings( One_String, Two_String );       {Illegal}
Two_String := 'ehcnalB olleH';
Test_Strings( 'ehcnalB olleH', Two_String );  {Legal}
```

In general, schema formal parameters provide more flexibility than conformant parameters. For instance, you can define undiscriminated formal parameters of record, set, or subrange types; you can provide corresponding

actual parameters with different discrimination values each time you call the routine. Consider the following:

```
TYPE
   Name_Type = STRING( 80 );
   Class_Grades( Num_Students, Num_Grades : INTEGER ) = RECORD
      Teacher        : Name_Type;
      Student_Names  : ARRAY[1..Num_Students] OF Name_Type;
      Student_Grades : ARRAY[1..Num_Grades] OF REAL;
      END;
VAR
   One_Class     : Class_Grades( 25, 10 );
   Two_Class     : Class_Grades( 32, 15 )
   Class_Average : REAL;

FUNCTION Get_Class_Average( Class : Class_Grades ) : REAL;
   {Function body...}

{In the executable section:}
Class_Average := Get_Class_Average( One_Class );
WRITELN( 'The grade average for this class is ', Class_Average );
Class_Average := Get_Class_Average( Two_Class );
WRITELN( 'The grade average for this class is ', Class_Average );
{...and so forth}
```

The function Get_Class_Average can process information for classes of any size and for classes that have any number of grades for each student.

In a few cases, conformant parameters can provide a better solution for a given application. For instance, when you want to define formal array parameters that accept variable-length actual parameters, you need to determine whether you want to restrict actual parameters to a schema family or if you want to accept actual parameters of a broader range of array data types. Consider the following:

```
TYPE
   My_Array( Upper : INTEGER ) = ARRAY[1..Upper] OF INTEGER;
   My_Subrange( Low, Up : INTEGER ) = Low..Up;
VAR
   i : My_Array( 10 );
   j : My_Array( 20 );
   k : ARRAY[ My_Subrange( 1, 10 ) ] OF INTEGER;
   l : ARRAY[ My_Subrange( 1, 10 ) ] OF INTEGER;
   m : ARRAY[ My_Subrange( 1, 20 ) ] OF INTEGER;

PROCEDURE Example1( p : My_Array ); {Body...}
PROCEDURE Example2( p : ARRAY[a..b : INTEGER] OF INTEGER ); {Body...}
```

The procedure Example1 accepts only variables of the type family My_Array (i and j). Variables k and l are not of the same type as variable i, despite being arrays of the same number of components of the same type; variables

k and l are also not of the same type as each other, since a separate ARRAY type specification indicates a unique type.

The procedure Example2 accepts all of the variables declared in the previous example.

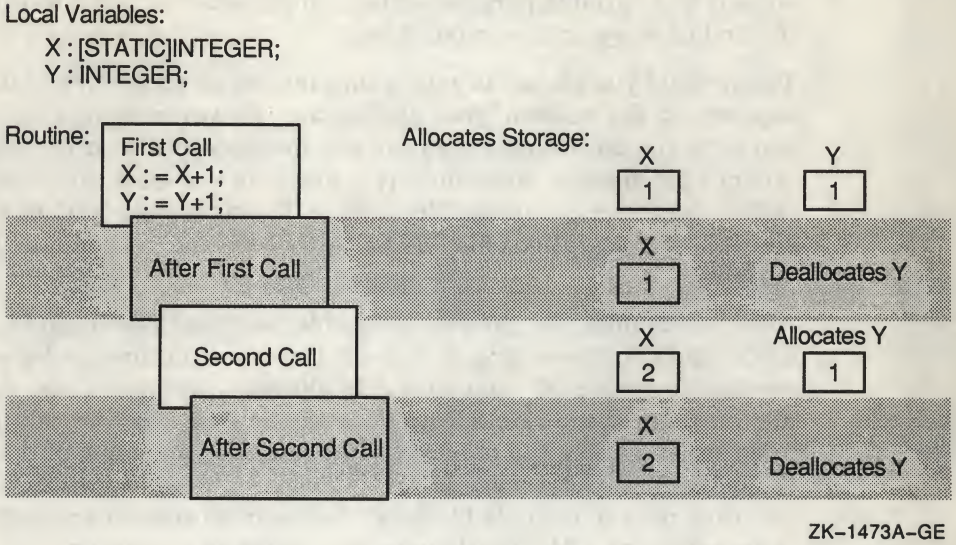
2.2 Static and Automatic Variable Allocation

The location of a declaration and the usage of the data determine the allocation of a variable. By default, with the exception of variables declared at the outermost level of a module, all variables are of the automatic allocation; variables at the outermost level of a module are of static allocation by default. If a variable is automatic, VAX Pascal allocates storage for the object every time control passes to the routine or block in which the variable is declared and deallocates storage every time control passes from the routine or block. If a variable is static, VAX Pascal allocates storage only once, and the storage exists as long as the containing executable image is active.

If you do not need to port your code to another vendor's compiler, you can use the VAX Pascal allocation attributes to control allocation of certain variables. (If you do need to port your code, you need to rely on default allocation of variables.) When working with routines, you can declare a static variable that is local to a routine; this allows you to use that variable's contents from one routine call to the next, since VAX Pascal does not deallocate the variable when control leaves the routine.

Figure 2–2 illustrates the effect of automatic and static variable allocation (this figure assumes that x and y are initialized to zero).

Figure 2–2: Static and Automatic Variable Allocation



The static variable x can keep a running total or incrementation from one routine call to the next; VAX Pascal deallocates y every time a routine finishes executing. Consider the function Get_Order from the larger program in Example 2–3:

```
FUNCTION Get_Order( VAR Total : REAL;
                    Done      : BOOLEAN ) : INTEGER;
   VAR
      Days_Total    : [STATIC] REAL VALUE 0.0; {Running total}
   BEGIN
   READLN( Item );
   CASE Item OF
      {Other case labels...}

      10 : Days_Total := Days_Total + Total;
      END; {CASE Item}

   IF Done AND ( Item <> 0 ) THEN   {With one customer}
      BEGIN
      {Write the customer's total for purchase...}
      Total := 0.0;
      END;
   Get_Order := Item;
   END; {Function Get_Item}
```

The variable Days_Total is static, so you can use it to keep a running total of all purchases during the day. The variable Total also keeps a running total (to add the price of each ordered item in successive recursive function calls), since it is a variable parameter. Get_Order sets the variable Total to zero at the end of every customer purchase.

The method you choose to retain information in successive routine calls depends on the needs of your application. However, using static variables can have the advantages of easier maintenance and security. From a maintenance perspective, sometimes it is easier to use local variables in routines rather than using variables declared at the outermost level of a program, since those declarations may be far away from the lines of code that use that variable.

Also, you could use a globally accessible variable (like Total) to record the day's totals, but according to scope rules, other routines and the program's executable section all have access to globally accessible data. Although allocation for the variable Days_Total exists from the first call to the end of program execution, the only code that can alter the value of the Days_Total is the function Get_Order. If your application needs to restrict access of sensitive data to a single block yet maintain storage in successive routine calls, using the STATIC attribute may be the best solution.

2.3 Structured Function-Return Values

Syntactically, a function call is an expression, since it returns a value. Consider the following code fragment from Example 2–3:

```
WHILE Get_Order( Total, Done ) <> 0 DO ;
```

The entire action of the statement is performed by the function call in the condition portion of the WHILE statement. When the function returns zero, the loop stops.

If a function returns a pointer, an array, or a record, the unextended Pascal standards do not allow you to dereference, index, or select a function call; these standards require you to assign the function-return value to a temporary variable and to dereference (or index or select) the variable.

The Extended Pascal standard lifts this restriction. Previously, you needed to use code like the following (which is based on declarations in Example 2–4):

```
Temp1 := Make_Node( 10 );
Temp2 := Make_Vector( 15 );

WRITELN( Temp1^.Data[1] );
WRITELN( Temp2[2] );
Temp1^.Data[1] := 42;
```

Using VAX Pascal, you can use the following code fragment from Example 2–4:

```
WRITELN( Make_Node( 10 )^.Data[1] );
WRITELN( Make_Vector( 15 )[2] );
Make_Node( 10 )^.Data[1] := 42;
```

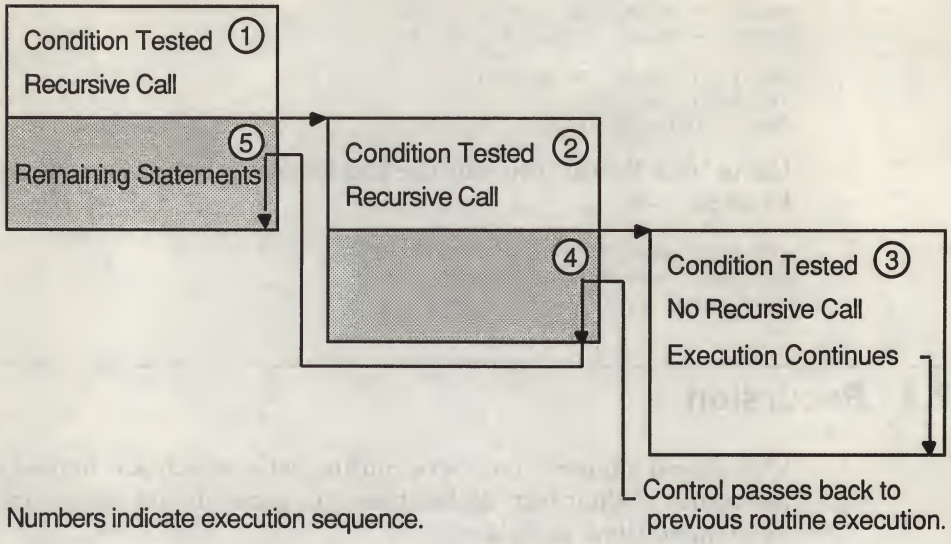
2.4 Recursion

VAX Pascal supports recursive routine calls, which are routine calls within the bodies of their own declarations. In general, you use recursion in your implementations as follows:

```
PROCEDURE Example( Param1, Param2 : INTEGER );
   BEGIN
   {Check a condition}

      Example( Int_Value_1, Int_Value_2 );

   {Else, allow completion of routine}
   END;
```

When you call a routine recursively, execution of the routine suspends at that point, and VAX Pascal begins another execution of that routine. When recursion ends and complete execution of a routine occurs, control passes back to the previous routine execution so that the remaining statements execute for that particular call, and so forth, until all remaining statements for all routine calls execute. Figure 2–3 illustrates the concept of recursion (the figure also assumes that x and y are initialized to zero).

Figure 2–3: Using Recursive Routine Calls

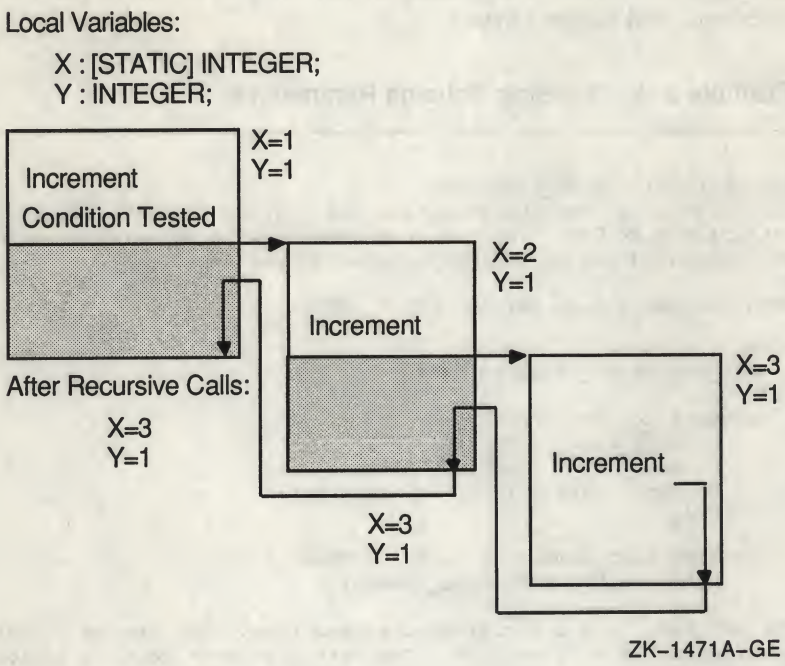


The code executes in the order shown by the numbers 1 through 5. The last code to execute is the statements following the first recursive call. When code in Part 5 executes, automatic variables have the same values that they did at the end of Part 1, not the same values as they did in Parts 2 through 4 (as each routine finishes execution, VAX Pascal deallocates storage for automatic variables).

If you want to maintain the storage of a local variable in successive recursive calls, you can declare the variable to have static allocation. (The variable Days_Total from Example 2–3 is declared as static.)

Figure 2–4 illustrates the lifetime of two variables (this figure assumes that x and y are initialized to zero).

Figure 2–4: Lifetime of Variables During Recursive Routine Calls



Within the code in the first routine execution after the recursive call, x has the value of 3, reflecting the incrementation that took place in recursive calls; variable y has the same value it did before the first recursive call.

2.5 Routine Examples

Example 2–1 shows the passing of parameters of undiscriminated and discriminated schema types.

Example 2–1: Passing Schema Parameters

```
{
Source File: SCHEMA_PARAMS.PAS
This example stores the names and the year of students listed in the
data file DATA.DAT.  This example shows how to declare parameters of
discriminated and of undiscriminated schema types.
}
PROGRAM Test_Schema_Params( INPUT, OUTPUT );

CONST
   Max_Number_of_Students = 5;
TYPE
   Student_Record = RECORD
      Student_Name  : STRING( 20 );
      Student_SS    : STRING( 10 );
      Student_Class : ( FR, SO, JR, SR );
      END;

   Student_List_Template( a, b : INTEGER ) =
      ARRAY[a..b] OF Student_Record;
   {
Student_List is a discriminated schema type.  The second actual
discriminant evaluates to 5.  The data file must contain 5 sets of
information.
   }
   Student_List = Student_List_Template( 1, Max_Number_of_Students );
```

(continued on next page)

Example 2–1 (Cont.): Passing Schema Parameters

```
VAR
   Students : Student_List;
{
Student_List_Template is undiscriminated.  The type of the
corresponding actual parameter must be derived from
Student_List_Template, but the actual discriminant can be of any
value.
}
PROCEDURE Get_Students( VAR Students : Student_List_Template );
   VAR
      Index         : INTEGER;
      Students_File : TEXT;
   BEGIN
   OPEN( FILE_NAME     := 'data.dat',
         FILE_VARIABLE := Students_File,
         HISTORY       := OLD );
   RESET( Students_File );
   FOR Index := Students.a TO Students.b   DO
      BEGIN
      READLN( Students_File, Students[Index].Student_Name );
      READLN( Students_File, Students[Index].Student_SS );
      READLN( Students_File, Students[Index].Student_Class );
      END;
   END; {PROCEDURE Get_Students}
{
Student_List is discriminated.  The corresponding actual parameter
must be of the Student_List_Template type, but the actual discriminant
of that type must equal the actual discriminant of the type of
Student_List (5).
}
PROCEDURE Write_Students( Students : Student_List );
   VAR
      Index : INTEGER;
   BEGIN
   FOR Index := Students.a TO Students.b   DO
      BEGIN
      WRITELN;
      WRITELN( Students[Index].Student_Name );
      WRITELN( Students[Index].Student_SS );
      WRITELN( Students[Index].Student_Class );
      END;
   END; {PROCEDURE Write_Students}
```

(continued on next page)

Example 2–1 (Cont.): Passing Schema Parameters

```
PROCEDURE Find_Class_Total( Students : Student_List );
   VAR
      Index, Fresh, Soph, Juni, Seni : INTEGER VALUE 0;
   BEGIN
   FOR Index := Students.a TO Students.b DO
      BEGIN
      CASE Students[Index].Student_Class OF
         FR : Fresh := Fresh + 1;
         SO : Soph  := Soph  + 1;
         JR : Juni  := Juni  + 1;
         SR : Seni  := Seni  + 1;
         OTHERWISE WRITELN( 'Student Without A Class' );
         END; {CASE Students}
   END;  {FOR Index}
   WRITELN;
   WRITELN( 'The number of freshman = ',   fresh );
   WRITELN( 'The number of sophomores = ', soph );
   WRITELN( 'The number of juniors  = ',   juni );
   WRITELN( 'The number of seniors  = ',   seni );
   END; {PROCEDURE Find_Class_Total}

BEGIN
Get_Students( Students );
Write_Students( Students );
Find_Class_Total( Students );
END.
{
Sample Input File DATA.DAT:

Urvashi Viad
009574560
FR
David Scondras
009342100
FR
Jeff Levi
006954532
SR
Eric Rofes
008794992
JR
Harry Britt
009564876
FR
}
```

Sample output from Example 2–1 is as follows:

```
Urvashi Viad
009574560
 FR

David Scondras
009342100
 FR

Jeff Levi
006954532
 SR

Eric Rofes
008794992
 JR

Harry Britt
009564876
 FR

The number of freshman   =          3
The number of sophomores =          0
The number of juniors    =          1
The number of seniors    =          1
```

Example 2–2 shows a Find_Node function that is functionally equivalent to Find_Node in Example 1–3, except that this routine is implemented using recursion.

Example 2–2: Recursive Function in a Binary Tree Program

```
{
Source File: BTREE_RECURSION.PAS
This function (Find_Node) locates a node in a binary tree and uses
recursive function calls.  The example is contrived to show simple,
recursive calls; you may wish to use more discretion when using
recursion for your applications.
}
FUNCTION Find_Node(
      Current : Ptr_Tree_Node;
      Name    : VARYING[Name_Bound] OF CHAR ) : Ptr_Tree_Node;
```

(continued on next page)

Example 2–2 (Cont.): Recursive Function in a Binary Tree Program

```
   BEGIN {Recursive implementation}
   IF Current = NIL THEN
      Find_Node := NIL
   ELSE
      IF Name = Current^.Name THEN
         Find_Node := Current
      ELSE
         BEGIN
         IF Name < Current^.Name THEN
            Find_Node := Find_Node( Current^.Left_Link, Name )
         ELSE Find_Node := Find_Node( Current^.Right_Link, Name );
         END; {ELSE of IF Name < Current^.Name}
   END; {of Find_Node}
```

Example 2–3 shows a program that reads orders from a customer at a fast-food restaurant, and provides a total for the purchase and a cash-register total for the day.

Example 2–3: Using Recursion to Generate Burger Orders

```
{
Source File: BURGERS.PAS
This program reads an order for burgers and fries, computes a total
for each customer (using recursive procedure calls), and computes a
running total for that cash register (using a STATIC variable).
}
PROGRAM Burger_Recursion( INPUT, OUTPUT );
VAR
   Total          : REAL VALUE 0.0;       {Customer's total}
   Done           : BOOLEAN VALUE TRUE;   {Done with 1 order}

PROCEDURE Print_Menu;
   BEGIN
   {
This procedure contains a series of WRITELN's that print the menu and
item numbers.  The user must type 10 to end an order and 0 to close
the register for the day.
   }
   END;
```

(continued on next page)

Example 2–3 (Cont.): Using Recursion to Generate Burger Orders

```
FUNCTION Get_Order( VAR Total : REAL;
                    Done      : BOOLEAN ) : INTEGER;
   VAR
      Item          : INTEGER;                  {Menu item    }
      Days_Total    : [STATIC] REAL VALUE 0.0; {Running total}
      Not_Done_Yet  : BOOLEAN VALUE FALSE;     {Still ordering}
   BEGIN
   READLN( Item );
   CASE Item OF
      0  : BEGIN
           WRITELN;
           WRITELN;
           WRITELN( 'The day''s total for this register is ',
                    Days_Total:4:2 );
           WRITELN( '=============================================' );
           END;

      1  : BEGIN
           Total := Total + 0.75;
           Get_Order( Total, Not_Done_Yet );
           WRITELN( ' Hamburger        .75' );
           END;

      2  : BEGIN
           Total := Total + 0.95;
           Get_Order( Total, Not_Done_Yet );
           WRITELN( ' Cheeseburger     .95' );
           END;

      3  : BEGIN
           Total := Total + 0.50;
           Get_Order( Total, Not_Done_Yet );
           WRITELN( ' Fries            .50' );
           END;

      4  : BEGIN
           Total := Total + 0.60;
           Get_Order( Total, Not_Done_Yet );
           WRITELN( ' Cola Soda        .60' );
           END;

      5  : BEGIN
           Total := Total + 0.60;
           Get_Order( Total, Not_Done_Yet );
           WRITELN( ' Orange Soda      .60' );
           END;

      10 : Days_Total := Days_Total + Total;
      END; {CASE Item}
```

(continued on next page)

Example 2–3 (Cont.): Using Recursion to Generate Burger Orders

```
   IF Done AND ( Item <> 0 ) THEN   {With one customer}
      BEGIN
      WRITELN('==========================');
      WRITELN( ' Your total is  ', Total:4:2 );
      WRITELN;
      WRITELN( 'Thank you.  Have a Burger Queen Day!' );
      WRITELN('========================================');
      WRITELN;
      WRITELN;
      Total := 0.0;
      END;
   Get_Order := Item;
   END; {Function Get_Item}

BEGIN
Print_Menu;
WHILE Get_Order( Total, Done ) <> 0 DO ;
END.
```

Sample output from Example 2–3 is as follows:

```
1
3
4
10
 Cola Soda         .60
 Fries             .50
 Hamburger         .75
==========================
 Your total is  1.85

Thank you.  Have a Burger Queen Day!
========================================


2
3
5
10
 Orange Soda       .60
 Fries             .50
 Cheeseburger      .95
==========================
 Your total is  2.05

Thank you.  Have a Burger Queen Day!
========================================

0

The day's total for this register is 3.90
================================================
```

Example 2–4 shows the dereferencing and indexing of function return values.

Example 2–4: Indexing, Dereferencing, and Selecting Function Results

```
{
Source File: FUNCTION_CALLS.PAS
This program dereferences, selects, and indexes function-return values.
}
PROGRAM Function_Calls( OUTPUT );
TYPE
   Vector = ARRAY[1..10] OF INTEGER;
   Ptr_To_Node = ^Node;
   Node = RECORD
      Data : Vector;
      Next : Ptr_To_Node;
      END;

FUNCTION Make_Vector( p : INTEGER ) : Vector;
   BEGIN          {Fill vector elements with passed parameter:}
   Make_Vector := Vector [OTHERWISE p];
   END;

FUNCTION Make_Node( p : INTEGER ) : Ptr_To_Node;
   VAR
      Ptr : Ptr_To_Node;
   BEGIN
   NEW( Ptr );
   Ptr^.Data := Make_Vector( p );
   Make_Node := Ptr;
   END;

BEGIN  {Program Function_Calls}
WRITELN( Make_Node( 10 )^.Data[1] );  {Dereference function result}
WRITELN( Make_Vector( 15 )[2] );      {Index function result}
{
Since pointer values can be used to reference variables,
you can dereference functions returning pointers on the left side of
an assignment statement.
}
Make_Node( 10 )^.Data[1] := 42;
END.
```

When you run the program in Example 2–4, you see the following output:

```
      10
      15
```

Chapter 3

Separate Compilation

VAX Pascal allows you to divide your application into subprograms by creating procedures and functions. VAX Pascal allows you further modularity by allowing you to create compilation units, called programs and modules, that can be compiled separately. This chapter discusses the following topics about separate compilation:

- The ENVIRONMENT, HIDDEN, and INHERIT attributes (Section 3.1)
- Interfaces and implementations (Section 3.2)
- A data model (Section 3.3)
- Examples (Section 3.4)

NOTE

The sections at the beginning of this chapter use code fragments from the examples contained in the section at the end of the chapter. Complete code examples are located in Section 3.4.

For More Information:

- On the ENVIRONMENT, HIDDEN, and INHERIT attributes (*VAX Pascal Reference Manual*)
- On compiling and executing programs and modules (*VAX Pascal Reference Supplement for VMS Systems*)

3.1 The ENVIRONMENT, HIDDEN, and INHERIT Attributes

To divide your program into a program and a series of modules, you need to decide, according to the needs of your application, which data types, constants, variables, and routines need to be shared either by other modules or by the program. To share data, create an environment file by using the ENVIRONMENT attribute in a module. Consider the following:

```
{
Source File: SHARE_DATA.PAS
This program initializes data to be shared with another compilation
unit.
}
[ENVIRONMENT( 'share_data' )]
Module Share_Data;
CONST
   Rate_For_Q1 = 0.1211;
   Rate_For_Q2 = 0.1156;
   Rate_For_Q3 = 0.1097;
   Rate_For_Q4 = 0.11243;
TYPE
   Initialized_Type = ARRAY[1..10] OF INTEGER VALUE
                      [1..5: 67; 6,9: 105; OTHERWISE 33];
END.
```

If you do not specify a file name, VAX Pascal creates an environment file using the file name of the source file and a default extension of .PEN. Another compilation unit can access the types and constants in the previous example by inheriting the environment file, as follows:

```
{
Source File: PROGRAM.PAS
This code inherits data declarations and uses them in a program.
}
[INHERIT( 'share_data' )]
PROGRAM Use_Data( OUTPUT );
VAR
   a, b, c          : Initialized_Type;
   Total            : REAL VALUE 0.0;
BEGIN
Total := Total + ( Total * Rate_For_Q3 );
WRITELN( b[7] );   {b is of an initialized type}
END.
```

To build the application made up of the code in the previous examples, type the following:

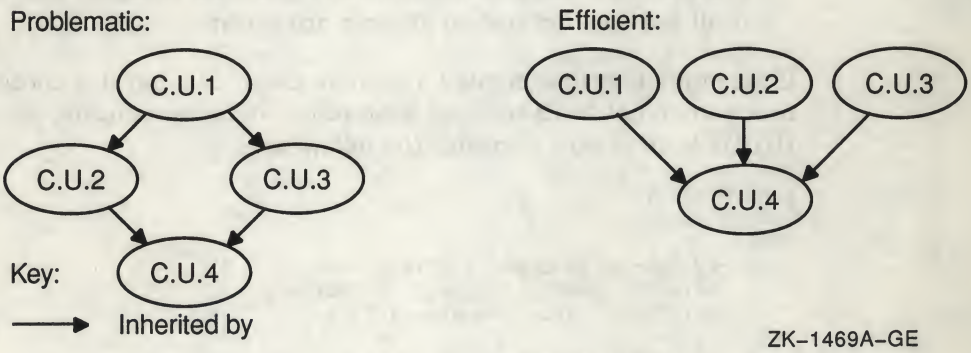
```
$ PASCAL SHARE_DATA
$ PASCAL PROGRAM
$ LINK PROGRAM
$ RUN PROGRAM
    33
```

If a module contains variable declarations, routine declarations, schema types, or module initialization or finalization sections, you must link the program with the module that created the environment file to resolve external references. To prevent errors, you may wish to link programs with modules of inherited environment files as standard programming practice. For instance, if SHARE_DATA contained a variable declaration, you must type the following to resolve the external reference:

```
$ PASCAL SHARE_DATA
$ PASCAL PROGRAM
$ LINK PROGRAM, SHARE_DATA
$ RUN PROGRAM
    33
```

For many applications, it is a good idea to place all globally accessible data into one module, create a single environment file, and inherit that module in other compilation units that need to make use of that data. Using environment files in this way reduces the difficulties in maintaining the data (it is easier to maintain one file) and it eliminates problems that can occur when you **cascade** environment files. If compilation unit A inherits an environment file from compilation unit B, and if unit B inherits a file from unit C, then inheritance is cascading. Figure 3–1 shows a cascading inheritance path and a noncascading inheritance path.

Figure 3–1: Cascading Inheritance of Environment Files



Cascading is not always undesireable; it depends on your application and on the nature of the environment files. For instance, if cascading occurs for a series of constant and type definitions that are not likely to change, cascading may require very little recompiling and relinking. However, if the

constant and type definitions change often or if environment files contain routines and variables, you may find it easier to redesign the inheritance paths of environment files due to the recompiling and relinking involved.

Also, the inheritance path labeled "efficient" in Figure 3–1 is not immune to misuse. That inheritance path, although it avoids the problems of cascading, may still involve multiply-declared identfiers (identical identifiers contained in several of the compilation units whose environment files are inherited by compilation unit 4).

In many instances, VAX Pascal does not allow multiply-declared identifiers in one application. For instance, a compilation unit cannot inherit two environment files that declare the same identifier; also, a compilation unit usually cannot inherit an environment file that contains an identifier that is identical to an identifier in the outermost level of the unit (one exception, for instance, is the redeclaration of a redefinable reserved word or of an identifier predeclared by VAX Pascal). Also, VAX Pascal allows the following exceptions to the rules concerning multiply-declared identifiers:

- A variable identifier can be multiply declared if all declarations of the variable have the same type and attributes, and if all but one declaration at most are external.
- A procedure identifier can be multiply declared if all declarations of the procedure have congruent parameter lists and if all but one declaration at most are external.
- A function identifier can be multiply declared if all declarations of the function have congruent parameter lists and identical result types, and if all but one declaration at most are external.

If a compilation unit creates an environment file and if it contains data that you do not want to share with other compilation units, you can use the HIDDEN attribute. Consider the following:

```
[ENVIRONMENT]
MODULE Example;
TYPE
   Array_Template( Upper : INTEGER ) =
      [HIDDEN] ARRAY[1..Upper] OF INTEGER;
   Global_Type : Array_Template( 10 );
VAR
   i : [HIDDEN] INTEGER;   {Used for local incrementing}

PROCEDURE x;
   BEGIN
      i := i + 1;
   END;
```

```
PROCDURE y;
   BEGIN
      FOR i := i + 1;
   END;
END.
```

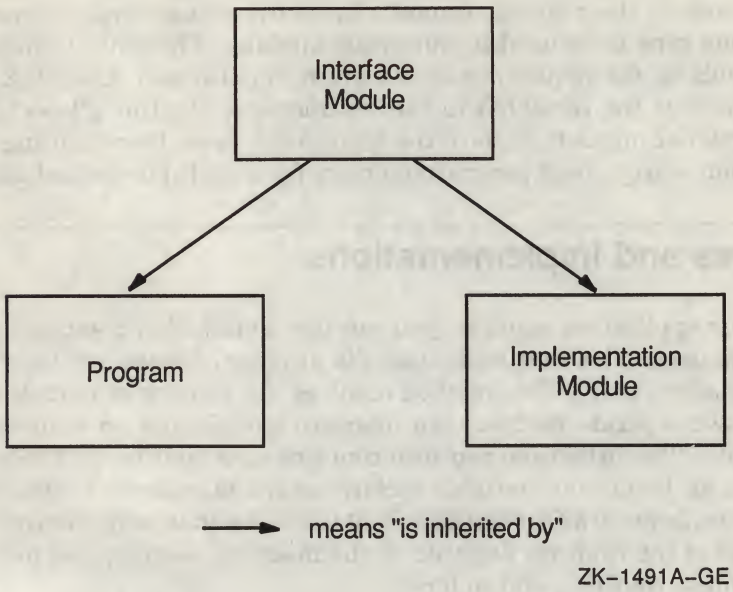
The code in the previous example hides the schema type, preventing the schema type to be used in inheriting modules. (Whether to hide the type depends on the requirements of a given application.) Also, VAX Pascal does not include the variable i in the environment file; this allows inheriting compilation units to declare the identifier i as an incrementing variable without worry about generating errors for a multiply-defined identifier.

3.2 Interfaces and Implementations

If your application requires, you can use a method of creating and inheriting environment files that minimizes the number of times you have to recompile compilation units. This method involves the division of module declarations into two separate modules: an interface module and an implementation module. The **interface module** contains data that is not likely to change: constant definitions, variable declarations, and external routine declarations. The **implementation module** contains data that may change: routine bodies of the routines declared in the interface module, and private types, variables, routines, and so forth.

The interface module creates the environment file that is inherited by both the implementation module and by the program. Figure 3–2 illustrates the inheritance process.

Figure 3–2: Inheritance Path of an Interface, an Implementation, and a Program



Consider this code fragment from Example 3–1:

```
[ENVIRONMENT( 'interface' )]
MODULE Graphics_Interface( OUTPUT );

   {Globally accessible type}

{Provide routines that manipulate the shapes:}
PROCEDURE Draw( s : Shape ); EXTERNAL;
PROCEDURE Rotate( s : Shape ); EXTERNAL;
PROCEDURE Scale( s : Shape ); EXTERNAL;
PROCEDURE Delete( s : Shape ); EXTERNAL;

   {Module initialization section}

END.
```

The code contained in the interface is not likely to change often. The implementation code can change without requiring recompilation of the other modules in the application. Consider this code fragment from Example 3–2:

```
[INHERIT( 'Interface' )]   {Predeclared graphics types and routines}
MODULE Graphics_Implementation( OUTPUT );

[GLOBAL] PROCEDURE Rotate( s : Shape );
   BEGIN
   WRITELN( 'Rotating the shape :', s.t );
   END;
```

To compile, to link, and to run the code in Examples 3–1, 3–2, and Example 3–3, type the following:

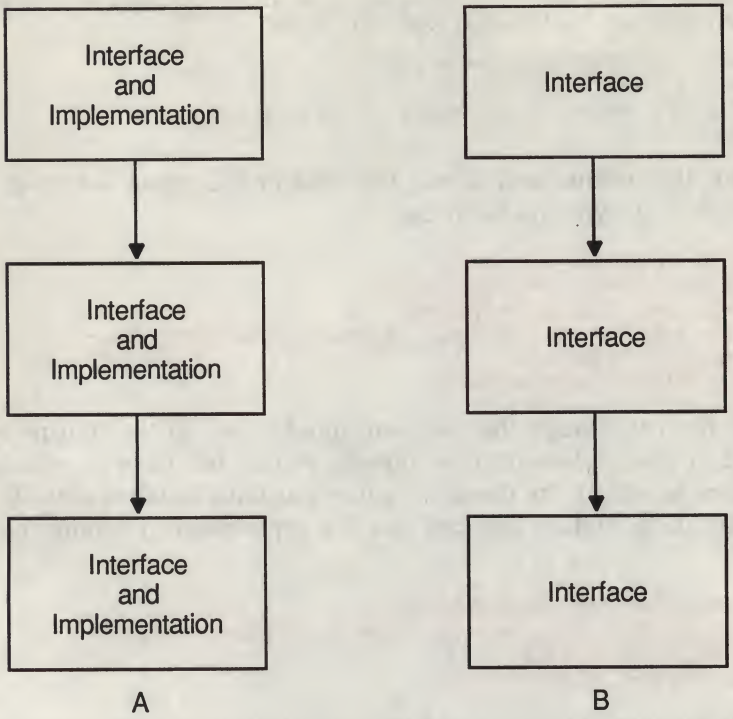
```
$  PASCAL GRAPHICS_INTERFACE
$  PASCAL GRAPHICS_IMPLEMENTATION
$  PASCAL GRAPHICS_MAIN_PROGRAM
$  LINK GRAPHICS_MAIN_PROGRAM, GRAPHICS_IMPLEMENTATION,-
_$  GRAPHICS_INTERFACE
$  RUN GRAPHICS_MAIN_PROGRAM
```

If you need to change the code contained in any of the routine bodies declared in the implementation module, you do not have to recompile the program to reflect the changes. After you have finished edits to the implementation module, you can run the application by typing the following code:

```
$  PASCAL GRAPHICS_IMPLEMENTATION
$  LINK GRAPHICS_MAIN_PROGRAM, GRAPHICS_IMPLEMENTATION,-
_$  GRAPHICS_INTERFACE
$  RUN GRAPHICS_MAIN_PROGRAM
```

In this manner, interfaces and implementations can save you maintenance time and effort. In addition, the interface and implementation design allows you to better predict when cascading inheritance may provide maintenance problems. Figure 3–3 illustrates two forms of cascading.

Figure 3–3: Cascading Using the Interface and Implementation Design



If the compilation units creating environment files are designed to contain both interface and implementation declarations, the cascading in column A may lead to more recompiling, more relinking, and more multiply-declared identifiers. The design illustrated by column B does not always provide easy maintenance, but it is more likely to do so. For instance, if each interface provided a different kind of constant or type (as determined by your application) and if the constants and types are not derived from one another, the inheritance path in column B may be quite efficient and orderly, and may require little recompiling and relinking.

As a restriction, you should not place the following in an implementation module:

- Nonstatic types and variables at the module level
- A module initialization section (TO BEGIN DO)
- A module finalization section (TO END DO)

Using these objects in implementation modules is discouraged because VAX Pascal cannot determine the order of activation of the initialization and finalization sections that do not directly follow an environment-file inheritance path. Since implementation modules do not create environment files, the initialization and finalization sections in those modules are effectively outside of any inheritance path. Also, if you use the objects listed previously in implementation modules, there may be attempts to access data that has not yet been declared. For instance, consider the following:

```
{In one file:}
[ENVIRONMENT( 'interface' )]
MODULE Interface;
PROCEDURE x; EXTERNAL;
END.

{In another file:}
[INHERIT( 'interface' )]
MODULE Implementation( OUTPUT );
VAR
   My_String : STRING( 10 );

[GLOBAL] PROCEDURE x;
   BEGIN
   WRITELN( My_String );
   END;

TO BEGIN DO
   My_String := 'Okay';
END.
```

In the previous example, it is possible for you to call procedure x (in some other module that also inherits INTERFACE.PEN) before the creation and initialization of variable My_String. You can circumvent this problem by using a routine call to initialize the variable and by moving the code to the interface module. Consider the following:

```
{In one file:}
[ENVIRONMENT( 'interface' )]
MODULE Interface;
VAR
   My_String : STRING( 10 );

PROCEDURE x; EXTERNAL;
PROCEDURE Initialize; EXTERNAL;
```

```
TO BEGIN DO
   Initialize;
END.

{In another file:}
[INHERIT( 'interface' )]
MODULE Implementation( OUTPUT );

[GLOBAL] PROCEDURE x;
   BEGIN
   WRITELN( My_String );
   END;

[GLOBAL] PROCEDURE Initialize;
   BEGIN
   My_String := 'Okay';
   END;
END.
```

3.3 Data Models

Using separate compilation and a few other features of VAX Pascal (including initial states, constructors, the HIDDEN attribute, and TO BEGIN DO and TO END DO sections), you can construct models for creating, distributing, isolating, and restricting data in an application.

Of course, the design of the data model depends on the needs of a particular application. However, to illustrate some of the power of VAX Pascal features used in conjunction, Examples 3–1, 3–2, and 3–3 create a generic graphics application. Consider the following code fragment from Example 3–1:

```
TYPE
   Shape_Types = ( Rectangle, Circle ); {Types of graphics objects}

   Shape( t : Shape_Types ) = RECORD
                                  {Starting coordinate points}
      Coordinate_X, Coordinate_Y : REAL VALUE 50.0;
      CASE t OF                       {Shape-specific values}
         Rectangle : ( Height, Width : REAL VALUE 10.0 );
         Circle    : ( Radius        : REAL VALUE 5.0 );
      END;

{Provide routines that manipulate the shapes:}
PROCEDURE Draw( s : Shape ); EXTERNAL;
PROCEDURE Rotate( s : Shape ); EXTERNAL;
PROCEDURE Scale( s : Shape ); EXTERNAL;
PROCEDURE Delete( s : Shape ); EXTERNAL;
```

The interface module provides an interface to the rest of the application. This module contains types and external procedure declarations that the data model choses to make available to other compilation units in the application; other units can access these types and routines by inheriting the generated environment file.

The type Shape_Types defines two legal graphical objects for this application: a circle and a rectangle. The type Shape can be used by other units to create circles and rectangles of specified dimensions. This code uses a variant record to specify the different kinds of data needed for a circle (a radius value) and a rectangle (height and width values).

Since the type has initial-state values, any variable declared to be of this type receives these values upon declaration. Providing initial states for types that are included in environment files can prevent errors when other compilation units try to access uninitialized data.

The initial states in this code are specified for the individual record values. You can also provide an initial state for this type using a constructor, as follows:

```
Shape( t : Shape_Types ) = RECORD
   Coordinate_X, Coordinate_Y : REAL;
   CASE t OF
      Square : ( Height, Width : REAL );
      Circle : ( Radius        : REAL );
   END VALUE [ Coordinate_X : 50.0; Coordinate_Y : 50.0;
               CASE Circle OF [ Radius : 5.0 ] ];
```

If you use constructors for variant records, you can only specify an initial state for one of the variant values. If you need to specify initial states for all variant values, you must specify the initial states on the individual variants, as shown in Example 3–1.

The interface module also declares routines that can draw, rotate, scale, and delete an object of type Shape. The bodies of these routines are located in the implementation module. The interface module also contains a TO BEGIN DO section, as shown in the following code fragment:

```
[HIDDEN] PROCEDURE Draw_Logo; EXTERNAL;

{
Before program execution, display a logo to which the main
program has no access.
}
TO BEGIN DO
   Draw_Logo;
```

As with the other routines, the body of Draw_Logo is located in the implementation module. The HIDDEN attribute prevents compilation units that inherit the interface environment file from calling the Draw_Logo routine. This ensures that the application only calls Draw_Logo once, at the beginning of the application.

Using this design, the interface module can provide graphical data and tools to be used by other compilation units without the other units having to worry about implementation details. The actual details are contained in one implementation module. For instance, the routine bodies are contained in the implementation module. Consider the following code fragment from Example 3–2:

```
{Declare routine bodies for declarations in the interface}
[GLOBAL] PROCEDURE Draw( s : Shape );
   BEGIN
   CASE s.t OF
      Circle    : WRITELN( 'Code that draws a circle' );
      Rectangle : WRITELN( 'Code that draws a rectangle' );
      END;
   END; {Procedure Draw}
```

The routine bodies of the external routines declared in the interface module are located in the implementation module. The code in each of the routines uses the actual discriminant of parameter s to determine if the shape is a circle or a rectangle and draws the shape. If this code needs to change, it does not require that you recompile the code in Examples 3–1 or 3–3.

Example 3–2 also contains code that is isolated and hidden from other compilation units that inherit the interface environment file. Consider the following code fragment from the interface module:

```
[GLOBAL] PROCEDURE Draw_Logo;
   VAR
      Initial_Shape  : Shape( Circle ) {Declare object}
         VALUE [ Coordinate_X : 50.0;
                 Coordinate_Y : 50.0;
                 CASE Circle OF
                 [Radius      : 15.75;]];
   BEGIN
   WRITELN( 'Drawing a company logo' );
   Draw( Initial_Shape );
   {Code pauses for 30 seconds as the user looks at the logo}
   Delete( Initial_Shape );
   WRITELN;
   {Ready for the rest of the graphics program to begin}
   END;
```

In the graphical data model, you may wish to define a company logo, and you may wish to display that logo on the screen before any other graphical data is drawn or displayed. This code declares the variable Initial_Shape. Since this variable is declared locally to Draw_Logo and since Draw_Logo is contained in a module that does not produce an environment file, other modules that may have access to the interface environment file do not have access to this variable. In this application, you may not wish to give other compilation units the power to alter the company logo.

The code in the interface's TO BEGIN DO, which executes before any program code, displays the company logo and deletes it to begin the application. Consider again the compilation process for interfaces, implementations, and programs:

```
$ PASCAL GRAPHICS_INTERFACE
$ PASCAL GRAPHICS_IMPLEMENTATION
$ PASCAL GRAPHICS_MAIN_PROGRAM
$ LINK GRAPHICS_MAIN_PROGRAM, GRAPHICS_IMPLEMENTATION,-
_$  GRAPHICS_INTERFACE
$ RUN GRAPHICS_MAIN_PROGRAM
```

VAX Pascal executes the TO BEGIN DO section according to the inheritance order of environment files. Remember that VAX Pascal cannot determine the order of execution for TO BEGIN DO sections contained in implementation modules, so you should avoid using them there.

Using this design, you can allow different sites that run the graphics application to access global data through the interface module. One location can maintain and control the contents of the implementation module, shipping the implementation's object module for use at other sites. This method can be used for other types of sensitive data or data that needs to be maintained locally.

3.4 Separate-Compilation Examples

Example 3–1 shows an interface module that creates the environment file INTERFACE.PEN. This environment file is inherited in Examples 3–2 and in 3–3.

Example 3–1: An Interface Module for Graphics Objects and Routines

```
{
Source File: GRAPHICS_INTERFACE.PAS
This module creates an interface to graphical data and routines.
}
[ENVIRONMENT( 'interface' )]
MODULE Graphics_Interface;
TYPE
   Shape_Types = ( Rectangle, Circle ); {Types of graphics objects}

   Shape( t : Shape_Types ) = RECORD
                                   {Starting coordinate points:}
      Coordinate_X, Coordinate_Y : REAL VALUE 50.0;
      CASE t OF                          {Shape-specific values}
         Rectangle : ( Height, Width : REAL VALUE 10.0 );
         Circle    : ( Radius        : REAL VALUE 5.0 );
      END;

{Provide routines that manipulate the shapes:}
PROCEDURE Draw( s : Shape ); EXTERNAL;
PROCEDURE Rotate( s : Shape ); EXTERNAL;
PROCEDURE Scale( s : Shape ); EXTERNAL;
PROCEDURE Delete( s : Shape ); EXTERNAL;
[HIDDEN] PROCEDURE Draw_Logo; EXTERNAL;

{
Before program execution, display a logo to which the main
program has no access.
}
TO BEGIN DO
   Draw_Logo;
END.
```

Example 3–2 shows the implementation of the routines declared in Example 3–1.

Example 3–2: An Implementation Module for Graphics Objects and Routines

```
{
Source File: GRAPHICS_IMPLEMENTATION.PAS
This module implements the graphics routines and data declarations
made global by the interface module.
}
[INHERIT( 'Interface' )]   {Predeclared graphics types and routines}
MODULE Graphics_Implementation( OUTPUT );

{Declare routine bodies for declarations in the interface:}
[GLOBAL] PROCEDURE Draw( s : Shape );
   BEGIN
   CASE s.t OF
      Circle    : WRITELN( 'Code that draws a circle' );
      Rectangle : WRITELN( 'Code that draws a rectangle' );
      END;
   END; {Procedure Draw}

[GLOBAL] PROCEDURE Rotate( s : Shape );
   BEGIN
   WRITELN( 'Rotating the shape :', s.t );
   END;

[GLOBAL] PROCEDURE Scale( s : Shape );
   BEGIN
   WRITELN( 'Scaling the shape :', s.t );
   END;

[GLOBAL] PROCEDURE Delete( s : Shape );
   BEGIN
   WRITELN( 'Deleting the shape :', s.t );
   END;
```

(continued on next page)

Example 3–2 (Cont.): An Implementation Module for Graphics Objects and Routines

```
[GLOBAL] PROCEDURE Draw_Logo;
   VAR
      Initial_Shape  : Shape( Circle ) {Declare object}
         VALUE [ Coordinate_X : 50.0;
                 Coordinate_Y : 50.0;
                 CASE Circle OF
                 [Radius       : 15.75;]];
   BEGIN
   WRITELN( 'Drawing a company logo' );
   Draw( Initial_Shape );
   {Code pauses for 30 seconds as the user looks at the logo}
   Delete( Initial_Shape );
   WRITELN;
   {Ready for the rest of the graphics program to begin}
   END;
END.
```

Example 3–3 shows a main program and its use of the types and routines provided by the interface module.

Example 3–3: A Graphics Main Program

```
{
Source File: GRAPHICS_MAIN_PROGRAM.PAS
This program inherits the interface environment file, which gives it
access to the implementation's declarations.
}
[INHERIT( 'Interface' )]  {Types and routines in Interface module}
PROGRAM Graphics_Main_Program( OUTPUT );

VAR
   My_Shape : Shape( Rectangle )
      VALUE [ Coordinate_X : 25.0;
              Coordinate_Y : 25.0;
              CASE Rectangle OF
              [Height : 12.50; Width : 25.63]];
BEGIN
{
You cannot access the variable Initial_Shape, because it is in  the
implementation module, and that module does not create an environment
file.
```

(continued on next page)

Example 3–3 (Cont.): A Graphics Main Program

```
You can work with My_Shape.  If you did not provide initial values in
this declaration section, the module Graphics_Interface provided
initial values for the schema type Shape.
} .
Draw(   My_Shape );
Scale(  My_Shape );
Rotate( My_Shape );
Delete( My_Shape );
END.
```

To compile, to link, and to run the code in Examples 3–1, 3–2, and Example 3–3, type the following:

```
$  PASCAL GRAPHICS_INTERFACE
$  PASCAL GRAPHICS_IMPLEMENTATION
$  PASCAL GRAPHICS_MAIN_PROGRAM
$  LINK GRAPHICS_MAIN_PROGRAM, GRAPHICS_IMPLEMENTATION,-
_$  GRAPHICS_INTERFACE
$  RUN GRAPHICS_MAIN_PROGRAM
Drawing a company logo
Code that draws a circle
Deleting the shape :    CIRCLE

Code that draws a rectangle
Scaling the shape : RECTANGLE
Rotating the shape : RECTANGLE
Deleting the shape : RECTANGLE
```

Chapter 4

Program Optimization and Efficiency

The objective of **optimization** is to produce source and object programs that achieve the greatest amount of processing with the least amount of time and memory.

Improved program efficiency results when programs are carefully designed and written, and when compilation techniques take advantage of your operating system and machine architecture environment. (The benefits of portable code and program efficiency depend on the requirements of your application.) The VAX Pascal compiler produces efficient code by utilizing the benefits of the system and architecture environment you use. The primary goal of optimization performed by the compiler is faster program execution.

This chapter discusses the following topics:

- Compiler optimizations (Section 4.1)
- Programming considerations (Section 4.2)
- Optimization considerations (Section 4.3)

4.1 Compiler Optimizations

Programs compiled with the VAX Pascal compiler undergo the process known as optimization by default. An optimizing compiler automatically attempts to remove repetitious instructions and redundant computations by making assumptions about the values of certain variables. This, in turn, reduces the size of the object code, allowing a program written in a high-level language to execute at a speed comparable to that of a well-written assembly language program. Although optimization can increase the amount of time required to compile a program, it results in a program that may execute faster and more efficiently than a nonoptimized program.

The language elements you use in the source program directly affect the compiler's ability to optimize the object program. Therefore, you should be aware of the ways in which you can assist compiler optimization. In addition, this awareness often makes it easier for you to track down the source of a problem when your program exhibits unexpected behavior.

The compiler performs the following optimizations:

- Compile-time evaluation of constant expressions
- Elimination of some common subexpressions
- Partial elimination of unreachable code
- Code hoisting from structured statements, including the removal of invariant computations from loops
- Inline code expansion for many predeclared functions
- Inline code expansion for user-declared routines
- Rearrangement of unary minus and NOT operations to eliminate unary negation and complement operations
- Partial evaluation of logical expressions
- Propagation of compile-time known values

These optimizations are described in the following sections. In addition, the compiler performs the following optimizations, which can be detected only by a careful examination of the machine code produced by the compiler.

- Global assignment of variables to registers

  If possible, reduce the number of memory references needed by assigning frequently referenced variables to registers.
- Reordering the evaluation of expressions

  This minimizes the number of temporary values required.
- Peephole optimization of instruction sequences

  The compiler examines code a few instructions at a time to find operations that can be replaced by shorter and faster equivalent operations.

For More Information:

For information on VAX Pascal language elements, see the *VAX Pascal Reference Manual*.

4.1.1 Compile-Time Evaluation of Constants

The compiler performs the following computations on constant expressions at compile time:

- Negation of constants

  The value of a constant preceded by unary minus signs is negated at compile time. For example:

  ```
  x := -10.0;
  ```

  This is compiled as a single instruction.

- Type conversion of constants

  The value of a lower-ranked constant is converted to its equivalent in the data type of the higher-ranked operand at compile time. For example:

  ```
  x := 10 * y;
  ```

  If x and y are both real numbers, then this operation is compiled as follows:

  ```
  x := 10.0 * y;
  ```

- Arithmetic on integer and real constants

  An expression that involves +, –, *, or / operators is evaluated at compile time. For example:

  ```
  CONST
     nn = 27;
  {In the executable section:}
  i := 2 * nn + j;
  ```

  This is compiled as follows:

  ```
  i := 54 + j;
  ```

- Array address calculations involving constant indexes

  These are simplified at compile time whenever possible. For example:

  ```
  VAR
     i : ARRAY[1..10, 1..10] OF INTEGER;
  {In the executable section:}
  i[1,2] := i[4,5];
  ```

  This is compiled as a single instruction.

- Evaluation of constant functions and operators

  Arithmetic, ordinal, transfer, unsigned, allocation size, CARD, EXPO, and ODD functions involving constants, concatenation of string constants, and logical and relational operations on constants, are evaluated at compile time.

For More Information:

For information on the complete list of compile-time operations and routines, see the *VAX Pascal Reference Manual*.

4.1.2 Elimination of Common Subexpressions

The same subexpression often appears in more than one computation within a program. For example:

```
a := b * c + e * f;

h := a + g - b * c;

IF ((b * c) - h) <> 0 THEN ...
```

In this code sequence, the subexpression b * c appears three times. If the values of the operands b and c do not change between computations, the value b * c can be computed once and the result can be used in place of the subexpression. Thus, the sequence shown above is compiled as follows:

```
t := b * c;

a := t + e * f;

h := a + g - t;

IF ((t) - h) <> 0 THEN ...
```

Two computations of b * c have been eliminated. In this case, you could have modified the source program itself for greater program optimization.

The following example shows a more significant application of this kind of compiler optimization, in which you could not reasonably modify the source code to achieve the same effect:

```
VAR
   a, b : ARRAY[1..25, 1..25] OF REAL;
{In the executable section:}
a[i,j] := b[i,j];
```

Without optimization, this source program would be compiled as follows:

```
t1 := (j - 1) * 25 + i;
t2 := (j - 1) * 25 + i;
a[t1] := b[t2];
```

Variables t1 and t2 represent equivalent expressions. The compiler eliminates this redundancy by producing the following optimization:

```
t = (j - 1) * 25 + i;
a[t] := b[t];
```

4.1.3 Elimination of Unreachable Code

The compiler can determine which lines of code, if any, are never executed and eliminates that code from the object module being produced. For example, consider the following lines from a program:

```
CONST
   Debug_Switch = FALSE;
{In the executable section:}
IF  Debug_Switch  THEN  WRITELN( 'Error found here' );
```

The IF statement is designed to write an error message if the value of the symbolic constant Debug_Switch is TRUE. Suppose that the error has been removed, and you change the definition of Debug_Switch to give it the value FALSE. When the program is recompiled, the compiler can determine that the THEN clause will never be executed because the IF condition is always FALSE; no machine code is generated for this clause. You need not remove the IF statement from the source program.

Code that is otherwise unreachable, but contains one or more labels, is not eliminated unless the GOTO statement and the label itself are located in the same block.

4.1.4 Code Hoisting from Structured Statements

The compiler can improve the execution speed and size of programs by removing invariant computations from structured statements. For example:

```
FOR j := 1 TO i + 23 DO
   BEGIN
   IF Selector THEN a[i + 23, j - 14] := 0
   ELSE b[i + 23, j - 14] := 1;
   END;
```

If the compiler detected this IF statement, it would recognize that, regardless of the Boolean value of Selector, a value is stored in the array

component denoted by [i + 23, j – 14]. Thus, the compiler would change the sequence to the following:

```
t := i + 23;
FOR j := 1 TO t DO
   BEGIN
   u := j - 14;
   IF Selector THEN a[t,u] := 0
   ELSE b[t,u] := 1;
   END;
```

This removes the calculation of j – 14 from the IF statement, and the calculation of i + 23 from both the IF statement and the loop.

4.1.5 Inline Code Expansion for Predeclared Functions

The compiler can often replace calls to predeclared routines with the actual algorithm for performing the calculation. For example:

```
Square := SQR( a );
```

The compiler replaces this function call with the following, and generates machine code based on the expanded call:

```
Square := a * a;
```

The program executes faster because the algorithm for the SQR function has already been included in the machine code.

4.1.6 Inline Code Expansion for User-Declared Routines

Inline code expansion for user-declared routines performs in the same manner as inline code expansion for predeclared functions: the compiler can often replace calls to user-declared routines with an inline expansion of the routine's executable code. Inline code expansion is useful on routines that are called only a few times. The overhead of an actual procedure call is avoided; therefore, program execution is faster. The size of the program, however, may increase due to the routine's expansion.

To determine whether or not it is desirable to inline expand a routine, the compiler uses a complex algorithm. The first part of the algorithm performs tests for cases that always prohibit the routine from being inlined. A failure of one of these tests can be thought of as a "hard failure." These hard failure

tests verify that the following are false, if any one of these tests is true, the routine is not inlined:

- The called routine is an external or inherited routine.
- Either the calling routine or the called routine does not have inlining optimization enabled. Note that optimization is enabled by default.
- The called routine establishes an exception handler, or is used as an exception handler.
- The called function result is a structured result type.
- The calling routine and the called routine do not have the same checking options enabled.
- The calling routine and the called routine do not use the same program section.
- The called routine declares a routine parameter or is itself a routine parameter.
- The called routine's parameter list contains a LIST or TRUNCATE parameter, a read-only VAR parameter, or a conformant parameter.
- The called routine declares local file variables or contains any nonlocal GOTO operations.
- The called routine references automatic variables in an enclosing scope.
- The called routine uses or declares nonstatic types.

The second part of the algorithm performs tests to determine how desirable it is to inline the routine at a particular call point. A failure to one of these tests can be thought of as a "soft failure." These tests check for the number of formal parameters, number of local variables, whether the called routine is directly recursive, the number of direct calls to the routine, and the size of both the calling and the called routine.

If an explicit [OPTIMIZE(INLINE)] attribute is specified on the routine declaration, the "hard failure" tests are still performed; however, the "soft failure" tests are not. So if the routine passes the "hard failure" tests, that routine is inlined at all call points. Specifying this attribute provides you with more power in deciding which routines should be inlined.

NOTE

There is no stack frame for an inline user-declared routine and no debugger symbol table information for the expanded routine. Debugging the execution of an inline routine is therefore difficult, and is not recommended.

4.1.7 Operation Rearrangement

The compiler can produce more efficient machine code by rearranging operations to avoid having to negate and then calculate the complement of the values involved. For example:

```
(-c) * (b - a)
```

If a program includes this operation, the compiler rearranges the operation to read as follows:

```
c * (a - b)
```

These two operations produce the same result, but because the compiler has eliminated negation or complement operations, the machine code produced is more efficient.

4.1.8 Partial Evaluation of Logical Expressions

The Pascal language does not specify the order in which the components of an expression must be evaluated. If the value of an expression can be determined by partial evaluation, then some subexpressions may not be evaluated at all. This situation occurs most frequently in the evaluation of logical expressions. For example:

```
WHILE ( i < 10 ) AND ( a[i] <> 0 ) DO
   BEGIN
   a[i] := a[i] + 1;
   i := i + 1;
   END;
```

In this WHILE statement, the order in which the two subexpressions (i < 10) and (a[i] <> 0) are evaluated is not specified; in fact, the compiler may evaluate them simultaneously. Regardless of which subexpression is evaluated first, if its value is FALSE, the condition being tested in the WHILE statement is also FALSE. The other subexpression need not be evaluated at all. Thus, in this case, the body of the loop is never executed.

To force the compiler to evaluate expressions in left-to-right order with short circuiting, you can use the AND_THEN operator, as in the following example:

```
WHILE ( i < 10 ) AND_THEN ( a[i] <> 0 ) DO
   BEGIN
   a[i] := a[i] + 1;
   i := i + 1;
   END;
```

4.1.9 Value Propagation

The compiler keeps track of the values assigned to variables and traces the values to most of the places that they are used. If it is more efficient to use the value rather than a reference to the variable, the compiler makes this change. This optimization is called value propagation. Value propagation causes the object code to be smaller, and may also improve run-time speed.

Value propagation performs the following actions:

- It allows run-time operations to be replaced with compile-time operations. For example:

```
Pi := 3.14;
Pi_Over_2 := Pi/2;
```

  In a program that includes these assignments, the compiler recognizes the fact that Pi's value did not change between the time of Pi's assignment and its use. So, the compiler would use Pi's value instead of a reference to Pi and perform the division at compile time. The compiler treats the assignments as if they were as follows:

```
Pi := 3.14;
Pi_Over_2 := 1.57;
```

  This process is repeated, allowing for further constant propagation to occur.

- It allows comparisons and branches to be avoided at run time. For example:

```
x := 3;
IF x <> 3 THEN  y := 30
ELSE y := 20;
```

  In a program that includes these operations, the compiler recognizes that the value of x is 3 and the THEN statement cannot be reached. Thus, the compiler would generate code as if the statements were written as follows:

```
x := 3;
y := 20;
```

4.1.10 Alignment of Compiler-Generated Labels

The compiler aligns the labels it generates for the top of loops and the beginnings of ELSE branches on longword boundaries, filling in unused bytes with NOP instructions.

Using the VMS system on a VAX machine, a branch to a longword-aligned address is faster than a branch to an unaligned address. This optimization may increase the size of the generated code; however, it increases run-time speed.

4.1.11 Error Reduction Through Optimization

An optimized program produces results and run-time diagnostic messages identical to those produced by an equivalent unoptimized program. An optimized program may produce fewer run-time diagnostics, however, and the diagnostics may occur at different statements in the source program. For example:

Unoptimized Code	Optimized Code
a := x/y;	t := x/y;
b := x/y;	a := t;
FOR i := 1 TO 10 DO	b := t;
c[i] := c[i] * x/y;	FOR i := 1 TO 10 DO c[i] := c[i] * t;

If the value of y is 0.0, the unoptimized program produces 12 divide-by-zero errors at run time; the optimized program produces only one. (The variable t is a temporary variable created by the compiler.) Eliminating redundant calculations and removing invariant calculations from loops can affect the detection of such arithmetic errors. You should keep this in mind when you include error-detection routines in your program.

4.2 Programming Considerations

The language elements that you use in a source program directly affect the compiler's ability to optimize the resulting object program. Therefore, you should be aware of the following ways in which you can assist compiler optimization and thus obtain a more efficient program.

- Define constant identifiers to represent values that do not change during your program. The use of constant identifiers generally makes a program easier to read, understand, and modify later. In addition, the resulting object code is more efficient because symbolic constants are evaluated only once, at compile time, while variables must be reevaluated whenever they are assigned new values.
- Whenever possible, use the structured control statements CASE, FOR, IF-THEN-ELSE, REPEAT, WHILE, and WITH rather than the GOTO statement. Although the GOTO statement can be used to exit from a loop, careless use of it interferes with both optimization and the straightforward analysis of program flow.

NOTE

When both the REPEAT and WHILE statements are valid for a loop you should use the REPEAT statement because it allows for faster execution.

- Enclose in parentheses any subexpression that occurs frequently in your program. The compiler checks whether any assignments have affected the subexpression's value since its last occurrence. If the value has not changed, the compiler recognizes that a subexpression enclosed in parentheses has already been evaluated and does not repeat the evaluation. For example:

```
x := SIN( u + (b - c) );
y := COS( v + (b - c) );
```

The compiler evaluates the subexpression (b–c) as a result of performing the SIN function. When it is encountered again, the compiler checks to see whether new values have been assigned to either b or c since they were last used. If their values have not changed, the compiler does not reevaluate (b–c).

- Once your program has been completely debugged, disable all checking with [CHECK(NONE)] or with the appropriate compilation switch. Recall that VAX Pascal enables bounds and declaration checking by default. When no checking code is generated, more optimizations can occur, and the program executes faster.

  Integer overflow checking is disabled by default. If you are sure that your program is not in danger of integer overflow, you should not enable overflow checking. Because overflow checking precludes certain optimizations, you can achieve a more efficient program by leaving it disabled.

- When a variable is accessed by a program block other than the one in which it was declared, the variable should have static rather than automatic allocation. An automatically allocated variable has a varying location in memory; accessing it in another block is time-consuming and less efficient than accessing a static variable.
- Avoid using the same temporary variable many times in the course of a program. Instead, use a new variable every time your program needs a temporary variable. Because variables stored in registers are the easiest to access, your program is most efficient when as many variables as possible can be allocated in registers. If you use several different temporary variables, the lifetime of each one is greatly reduced; thus, there is a greater chance that storage for them can be allocated in registers rather than at memory locations.
- When creating schema records (or records with nonstatic fields), place the fields with run-time size at the end of the record. The generated code has to compute the offset of all record fields after a field with run-time size, and this change minimizes the overhead.

For More Information:

- On VAX Pascal language elements and on attributes (*VAX Pascal Reference Manual*)
- On compilation switches (*VAX Pascal Reference Supplement for VMS Systems*)

4.3 Optimization Considerations

Because the compiler must make certain assumptions in order to optimize a program, unexpected results may occur if you do not utilize the optimizations discussed in the following sections. If your program does not execute correctly because of undesired optimizations, you can use the NOOPTIMIZE attribute or the appropriate compilation switch to prevent optimizations from occurring.

For More Information:

- On attributes, and on static and automatic variables (*VAX Pascal Reference Manual*)
- On compilation switches (*VAX Pascal Reference Supplement for VMS Systems*)

4.3.1 Subexpression Evaluation

The compiler can evaluate subexpressions in any order and may even choose not to evaluate some of them. Consider the following subexpressions that involve a function with side effects:

```
IF f( a ) AND f( b ) THEN ...
```

This IF statement contains two designators for function f with the same parameter a. If f has side effects, the compiler does not guarantee the order in which the side effects will be produced. In fact, if one call to f returns FALSE, the other call to f might never be executed, and the side effects that result from that call would never be produced:

```
q := f( a ) + f( a );
```

The Pascal standard allows a compiler to optimize the code as follows:

```
Q := 2 * f( a )
```

If the compiler does so, and function f has side effects, the side effects would occur only once because the compiler has generated code that evaluates f(a) only once.

If you wish to ensure left-to-right evaluation with short circuiting, use the AND_THEN and OR_ELSE Boolean operators.

For More Information:

For information on the order of expression evaluation, see the description of the NOOPTIMIZE attribute in the *VAX Pascal Reference Manual.*

4.3.2 Lowest Negative Integer

The compiler assumes that all integer values are in the range –MAXINT through MAXINT. However, the VAX architecture supports an additional integer value, (–MAXINT–1). Should your program contain a subexpression with this value, its evaluation might result in an integer overflow trap. Therefore, a computation involving the value (–MAXINT–1) might not produce the expected result. To evaluate expressions that include (–MAXINT–1), you should disable either optimization or integer overflow checking.

4.3.3 Pointer References

The compiler assumes that the value of a pointer variable is either the constant identifier NIL or a reference to a variable allocated in heap storage by the NEW procedure. A variable allocated in heap storage is not declared in a VAR section and has no identifier of its own; you can refer to it only by the name of a pointer variable, followed by a circumflex (^). Consider the following:

```
VAR
   x : INTEGER;
   p : ^INTEGER;
{In the executable section:}
NEW( p );
p^ := 0;
x  := 0;
IF p^ = x THEN  p^ := p^ + 1;
```

If a pointer variable in your program must refer to a variable with an explicit name, that variable must be declared VOLATILE or READONLY. The compiler makes no assumptions about the value of volatile variables and therefore performs no optimizations on them.

Use of the ADDRESS function, which creates a pointer to a variable, can result in a warning message because of optimization characteristics. By passing a non-read-only or nonvolatile static or automatic variable as the parameter to the ADDRESS function, you indicate to the compiler that the variable was not allocated by NEW, but was declared with its own identifier. Because the compiler's assumptions are incorrect, a warning message occurs. You can also use IADDRESS, which functions similarly to the ADDRESS function, except that IADDRESS does not generate any warning messages. You should use caution when using IADDRESS.

Similarly, when the parameter to ADDRESS is a formal VAR parameter or a component of a formal VAR parameter, the compiler issues a warning message that not all dynamic variables allocated by NEW may be passed to the function.

For More Information:

For information on attributes and on predeclared routines, see the *VAX Pascal Reference Manual*.

4.3.4 Variant Records

Because all the variants of a record variable are stored in the same memory location, a program can use several different field identifiers to refer to the same storage space. However, only one variant is valid at a given time; all other variants are undefined. Thus, you must store a value in a field of a particular variant before you attempt to use it. For example:

```
VAR
   x : INTEGER;
   a : RECORD
       CASE t : BOOLEAN OF
          TRUE   : ( b : INTEGER );
          FALSE  : ( c : REAL );
       END;
{In the executable section:}
x := a.b + 5;
a.c := 3.0;
x := a.b + 5;
```

Record a has two variants, b and c, which are located at the same storage address. When the assignment a.c := 3.0 is executed, the value of a.b becomes undefined because TRUE is no longer the currently valid variant. When the statement x := a.b + 5 is executed for the second time, the value of a.b is unknown. The compiler may choose not to evaluate a.b a second time because it has retained the field's previous value. To eliminate any misinterpretations caused by this assumption, variable a should be associated with the VOLATILE attribute. The compiler makes no assumptions about the value of VOLATILE objects.

For More Information:

For information on variant records or on the VOLATILE attribute, see the *VAX Pascal Reference Manual.*

4.3.5 Type Cast Operations

When a type cast operation is performed, the compiler disregards any previous assumptions about the value of the cast object. This allows you to temporarily alter the type of the cast object at that point only. A type cast operation affects optimization only at the location in the program where the cast takes place. Optimizations elsewhere in the program and optimizations involving only uncast objects are not affected.

You should use type casts with care because the type cast operation can sometimes affect distant parts of a program. If a type cast on a variable is likely to affect its value at other points in the program, the variable should be declared VOLATILE.

For More Information:

For information on type casting or on the VOLATILE attribute, see the *VAX Pascal Reference Manual*.

4.3.6 Effects of Optimization on Debugging

Some of the effects of optimized programs on debugging are as follows:

- Use of registers

  When the compiler determines that the value of an expression does not change between two given occurrences, it may save the value in a register. In such a case, it does not recompute the value for the next occurrence, but assumes that the value saved in the register is valid. If, while debugging the program, you attempt to change the value of the variable in the expression, then the value of that variable is changed, but the corresponding value stored in the register is not. Thus, when execution continues, the value in the register may be used instead of the changed value in the expression, causing unexpected results.

  When the value of a variable is being held in a register, its value in memory is generally invalid; therefore, a spurious value may be displayed if you try to examine a variable under these circumstances.

- Coding order

  Some of the compiler optimizations cause code to be generated in a order different from the way it appears in the source. Sometimes code is eliminated altogether. This causes unexpected behavior when you try to step by line or use source display features.

- Use of condition codes

  This optimization technique takes advantage of the way in which the VAX processor condition codes are set. For example, consider the following source code:

```
x := x + 2.5;
IF x < 0  THEN ...
```

  Rather than test the new value of x to determine whether to branch, the optimized object code bases its decision on the condition code settings after 2.5 is added to x. Thus, if you attempt to set a debugging breakpoint at the second line and deposit a different value into x, you cannot achieve the intended result because the condition codes no longer reflect the value of x. In other words, the decision to branch is being made without regard to the deposited value of the variable.

- Inline code expansion on user-declared routines

  There is no stack frame for an inline user-declared routine and no debugger symbol table information for the expanded routine. Debugging the execution of an inline user-declared routine is therefore difficult, and is not recommended.

To prevent conflicts between optimization and debugging, you should always compile your program with a compilation switch that deactivates optimization until it is thoroughly debugged. Then you can recompile the program (which by default is optimized) to produce efficient code.

For More Information:

For information on debugging tools available in your environment and on compilation switches, see the *VAX Pascal Reference Supplement for VMS Systems*.

Chapter 5

Programming on VMS Systems

This chapter contains information on VAX Pascal features that are useful when using Record Management Services (RMS), the VMS Run-Time Library (RTL), and the VMS System Services. This chapter contains the following sections:

- Attributes (Section 5.1)
- Item lists (Section 5.2)
- Foreign mechanisms on actual parameters (Section 5.3)
- RMS examples (Section 5.4)
- RMS and RTL examples (Section 5.5)
- Condition handler examples (Section 5.6)
- System services examples (Section 5.7)
- DECwindows example (Section 5.8)

NOTE

The sections at the beginning of this chapter use code fragments from the examples contained in the sections at the end of the chapter. Complete code examples are located in Sections 5.4, 5.5, 5.6, and 5.7.

For More Information:

- On VAX Pascal language elements (*VAX Pascal Reference Manual*)
- On the VMS operating system (*VAX Pascal Reference Supplement for VMS Systems*)

5.1 Using Attributes

When writing programs that use VMS System Services and RTL routines, it is common to use several VAX Pascal attributes.

The VOLATILE attribute indicates that a variable is written or read indirectly without explicit program action. The most common occurrence for this is with item lists. In that case, the address of the variable is placed in the item list (most likely using the IADDRESS routine). This address is then used later when the entire item list is passed to a system service. Without the VOLATILE attribute, the compiler does not realize that the call to the system service or RTL routine uses the variable.

The UNBOUND attribute designates a routine that does not have a static link available to it. Without a static link, a routine can only access local variables, parameters, or statically allocated variables. System services that require AST or action routines want the address of an UNBOUND routine. Routines at the outer level of a PROGRAM or MODULE are UNBOUND by default.

The ASYNCHRONOUS attribute designates a routine that might be called asynchronously of any program action. This allows the compiler to verify that the asynchronous routine only accesses local variables, parameters, and VOLATILE variables declared at outer levels. Without the assurance that only VOLATILE variables are used, the asynchronous routine might access incorrect data, or data written by the routine will not be available to the main program.

5.2 Using Item Lists

Many VMS system services use item lists. **Item lists** are sequences of control structures that provide input to the system service and that describe where the service should place its output. These item lists can have an arbitrary number of cells and are terminated with a longword of value 0.

Since different programs need a different number of item list cells, a schema type can be used to define a generic item list data type. This schema type

can then be discriminated with the appropriate number of cells. Consider the following:

```
TYPE
   Item_List_Cell = RECORD
      CASE INTEGER OF
         1: (    { Normal Cell }
              Buffer_Length : [WORD] 0..65535;
              Item_Code     : [WORD] 0..65535;
              Buffer_Addr   : UNSIGNED;
              Return_Addr   : UNSIGNED
             );
         2: (    { Terminator }
              Terminator    : UNSIGNED
             );
      END;
   Item_List_Template( Count : INTEGER ) =
      ARRAY [1..Count] OF Item_List_Cell;
```

The Item_List_Cell data type specifies what a single cell looks like. The Buffer_Addr and Return_Addr fields are declared as UNSIGNED since most applications use the IADDRESS predeclared routine to fill them in. The Item_List_Template schema type defines an array of item list cells with a upper bound to be filled in by an actual discriminant.

To use this schema type, first determine the number of item list cells required including one cell for the terminator. After the number of cells has been determined, declare a variable discriminating the schema. Consider the following:

```
VAR
   Item_List : Item_List_Template( 2 );
```

Additionally, since actual discriminants to schema can be run-time expressions, you can write a routine that can have item lists with a number of cells that is determined at run time.

After the item list variable has been declared, each cell must be filled in according to the system service and operation requested.

Consider the following example using the SYS$TRNLNM system service:

```
VAR
   Item_List       : Item_List_Template( 2 );
   Translated_Name : [VOLATILE] VARYING [132] OF CHAR;

   {Specify the buffer to return the translation:}
   Item_List[1].Buffer_Length   := SIZE( Translated_Name.BODY );
   Item_List[1].Item_Code       := LNM$_String;
   Item_List[1].Buffer_Addr     := IADDRESS( Translated_Name.BODY );
   Item_List[1].Return_Addr     := IADDRESS( Translated_Name.LENGTH);

   { Terminate the item list:}
   Item_List[2].Terminator      := 0;
```

The VAR section declares an item list with two cells. It also declares an output buffer for the system service. The VOLATILE attribute is used since the call to SYS$TRNLNM indirectly writes into the variable. The first cell is filled in with the operation desired, the size of the output buffer, the location to write the result, and the location to write the size of the result.

Using the SIZE predeclared function prevents the code from having to be modified if the output buffer ever changes size. Using the BODY and LENGTH predeclared fields of the VARYING string allows the system service to construct a valid VARYING OF CHAR string. Finally, the second cell of the item list is initialized. Since the second cell is the last cell, the terminator field must be filled in with a value of 0.

5.3 Using Foreign Mechanism Specifiers on Actual Parameters

The definition files provided by VAX Pascal (SYS$LIBRARY:STARLET.PAS, and so forth) are created from a generic description language used by the VMS operating system. Since this description language does not contain all the features found in VAX Pascal, some of the translations do not take advantage of VAX Pascal features. Also since several of the system services are generic in nature, it is impossible to provide a definitive definition for every situation.

If a formal parameter definition does not reflect the current usage, you can use a foreign mechanism specifier to direct the compiler to use a different passing mechanism or different descriptor type than the default for that parameter.

Consider the following:

- The ASTADR parameter

  Many system services define this parameter to be a procedure parameter with no formal parameters. This is because the format of the arguments passed to the AST routine vary with the system service. If you specify a routine with parameters as the actual parameter to an ASTADR parameter, you will receive a compile-time error saying that the formal parameter and actual parameter have different parameter lists. To solve this problem, you can specify the %IMMED foreign mechanism specifier on the actual parameter. This causes the compiler to pass the address of the routine without verifying that the parameter lists are identical. •

- The ASTPRM parameter

  Many system services define this parameter to be an UNSIGNED parameter passed by immediate value. Since the parameter to an AST routine is dependent on the application, it is often desired to pass the address of a variable instead of its contents. To solve this problem, you can specify the %REF foreign mechanism specifier on the actual parameter. This causes the compiler to pass the address of the variable instead of the contents of the variable.

- The P1..Pn parameters

  The P1 through P6 parameters of the $QIO and $QIOW system services and the P1 through P20 parameters of the $FAO system services are also defined to be UNSIGNED parameters passed by immediate value. If the actual parameter is not UNSIGNED or requires a different passing mechanism, you can specify the %REF foreign mechanism specifier on the actual parameter. This causes the compiler to pass the address of the variable instead of the contents of the variable.

- The RESULTANT_FILESPEC parameter of the LIB$FIND_FILE Run-Time Library routine

  This parameter is declared to be a VAR conformant PACKED ARRAY OF CHAR parameter and is passed by CLASS_A descriptor. However, the LIB$FIND_FILE routine can also accept CLASS_VS descriptors of VARYING OF CHAR variables. To cause the compiler to build a CLASS_VS descriptor instead of the default CLASS_A descriptor, you can specify the %DESCR foreign mechanism specifier on the actual VARYING OF CHAR parameter.

5.4 RMS Examples

Example 5–1 provides an example of calling the RMS routine SYS$SETDDIR.

Example 5–1: Calling the RMS Routine SYS$SETDDIR

```
{
Source File: SYS$SETDDIR.PAS
The following program calls the RMS procedure SYS$SETDDIR to set the
default directory for a process.
}
[INHERIT( 'sys$library:starlet' )] PROGRAM Setd_Dir( OUTPUT );
TYPE
   Word_Integer = [WORD] 0..65535;
VAR
   Dir_Status : INTEGER;
   {
The $SETDIR routine accepts three parameters.  To omit the second and
third parameters, which are optional, supply default values in the
formal declaration of the routine.
   }

FUNCTION SYS$SETDDIR(
    New_Dir     : [CLASS_S] PACKED ARRAY[ 1..u : INTEGER ] OF CHAR;
    Old_Dir_Len : Word_Integer := %IMMED 0;
    Old_Dir     : VARYING[lim2] OF CHAR := %IMMED 0 ) : INTEGER;
                                                        EXTERNAL;
BEGIN   {Main program}

Dir_Status := SYS$SETDDIR( '[course.prog.pas]' );
IF NOT ODD( Dir_Status ) THEN
   WRITELN( 'Error in SYS$SETDDIR call.' );
END.    {Main program}
```

Example 5–2 shows an example of using the record file address (RFA) to access a file component (RMS refers to components as records).

Example 5–2: Using the RMS Record File Address (RFA)

```
{
Source File: RFA.PAS
This program reads a file, then prints it backwards.  This is done by
saving the RFAs as the file is read sequentially, then using RFAs to
step through the file backwards.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM RFA_Read( INPUT, OUTPUT );
CONST
   Max_Components = 1000;  {File cannot have > 1000 components}
TYPE                        {Components truncated at 132 characters:}
   File_Type       = FILE OF VARYING[132] OF CHAR;
   File_Name_Type  = VARYING[255] OF CHAR;
   Ptr_To_Fab      = ^FAB$TYPE;
   Ptr_To_Rab      = ^RAB$TYPE;
   RFA_Type = RECORD
      rfa0 : [LONG] UNSIGNED;
      rfa4 : [WORD] 0..65535;
      END;

[EXTERNAL] FUNCTION PAS$RAB(
          VAR f : [UNSAFE] File_Type ) : Ptr_To_Rab; EXTERNAL;

FUNCTION Position_Of( VAR f : File_Type ) : RFA_Type;
   {
This function returns the current position (the RFA) of the
Pascal file variable parameter.
   }
   VAR
      RAB : Ptr_To_RAB;
      RFA : RFA_Type;
   BEGIN
   RAB := PAS$RAB( f );
   RFA.rfa0 := RAB^.rab$l_rfa0;
   RFA.rfa4 := RAB^.rab$w_rfa4;
   Position_Of := RFA;
   END;
```

(continued on next page)

Example 5–2 (Cont.): Using the RMS Record File Address (RFA)

```
PROCEDURE Position_To( VAR f      : File_Type;
                      VAR rfa    : RFA_Type );
   {
This procedure positions a Pascal file variable to a given RFA.  It is
straight forward except for the RESET, which is needed to "reset" the
VAX Pascal internal structures when EOF is true.
   }
   VAR
      RAB     : Ptr_To_RAB;
      Status  : INTEGER;
   BEGIN
   IF EOF( f ) THEN
      RESET( f );                          {Reset EOF status}
   RAB := PAS$RAB( f );
   RAB^.rab$l_rfa0 := RFA.rfa0;            {Put RFA into RAB}
   RAB^.rab$w_rfa4 := RFA.rfa4;
   RAB^.rab$b_rac  := rab$c_rfa;           {Set for RFA access}

   Status := $GET( RAB^ );                 {Do the RMS get}
   IF NOT ODD( Status ) THEN
      LIB$STOP( Status );

   RAB^.rab$b_rac  := rab$c_seq;           {Restore to sequential access}

   f^.LENGTH := RAB^.rab$w_rsz;            {Update length of VARYING string
                                            usually done by VAX Pascal}
   END;

FUNCTION User_Open( VAR FAB : FAB$TYPE; VAR RAB : RAB$TYPE;
                    VAR f   : File_Type ) : INTEGER;
   {
This user-action routine modifies the FOP fields
to allow nonsequential access to the file.
   }
   VAR
      Status : INTEGER;
   BEGIN
   FAB.fab$v_sqo := FALSE;             {Allow nonsequential access}

   Status := $OPEN( FAB );             {Open file and connect RAB}
   IF ODD( Status ) THEN
      Status := $CONNECT( RAB );

   User_Open := Status;                {Return $OPEN or $CONNECT status}
   END;

VAR
    f           : FILE OF VARYING[132] OF CHAR;
    File_Name   : File_Name_Type;
    i, Line_Num : INTEGER;
    RFA_Array   : ARRAY[1..Max_Components] OF RFA_Type;
```

(continued on next page)

Example 5–2 (Cont.): Using the RMS Record File Address (RFA)

```
BEGIN
WRITE( 'file-name> ' );
IF NOT EOF THEN
   BEGIN
   {Get file-name and prepare for reading:}
   READLN( File_Name );
   OPEN( FILE_VARIABLE  := f,
         FILE_NAME      := File_Name,
         HISTORY        := READONLY,
         USER_ACTION    := User_Open );
   RESET( f );
   {
Read file saving RFAs in rfa_array.  Remember that the RESET has
already caused first record to be read into file buffer variable.
   }
   Line_Num := 0;
   WHILE NOT EOF( f ) DO
      BEGIN
      Line_Num := Line_Num + 1;
      RFA_Array[Line_Num] := Position_Of( f );
      GET( f );
      END;
   {Print file backwards:}
   FOR i := Line_Num DOWNTO 1 DO
      BEGIN
      Position_To( f, RFA_Array[i] );
      WRITELN( f^ );
      END;  {FOR i}
   END;  {IF NOT EOF}
END.
{
Sample input TEXT file DATA.DAT:

This is the first line.
This is the second line.
This is the third line.
}
```

The following is output when you run the program in Example 5–2:

```
$  RUN RFA
file-name>  TEMP.TXT
This is the third line.
This is the second line.
This is the first line.
```

From a VAX Pascal program, you can only create fixed-length indexed-sequential-access-method (ISAM) files. However, you can use a VAX Pascal program to access existing varying-length ISAM files. Example 5–3 shows an example of a source file in File Definition Language (FDL). If you enter

this code into a file and if you type the following DCL command, RMS creates a varying-length ISAM file:

```
$ CREATE/FDL=MAKE_V_ISAM.FDL
```

Example 5–3: FDL Code that Creates an ISAM File With Varying-Length Records

```
IDENT   "21-FEB-1987 16:08:16   VAX-11 FDL Editor"

FILE
        NAME                    "v.idx"
        ORGANIZATION            indexed

RECORD
        FORMAT                  variable

KEY 0
        SEG0_LENGTH             4
        SEG0_POSITION           0
        TYPE                    int4
```

The examples that follow contain code that reads from and writes to the ISAM file created by the FDL code in Example 5–3.

Example 5–4 shows how to write to the ISAM file with varying-length records.

Example 5–4: Writing to an ISAM File with Varying-Length Records

```
{
Source File: WRITE_V_ISAM.PAS
This example writes to an ISAM file with varying-length records.
}
PROGRAM idx_write;
TYPE
   rt = RECORD
      Int : INTEGER;
      Str : VARYING[132] OF CHAR;
      END;
   vt = VARYING[ SIZE( rt ) ] OF CHAR;
VAR
   My_String    : vt;
   My_Rec       : rt;
   ISAM_File    : FILE OF vt;
```

(continued on next page)

Example 5–4 (Cont.): Writing to an ISAM File with Varying-Length Records

```
BEGIN
OPEN( ISAM_File, 'v.idx', HISTORY := OLD, ACCESS_METHOD := KEYED,
      RECORD_TYPE := VARIABLE, ORGANIZATION := INDEXED );
EXTEND( ISAM_File );

My_Rec.Int          := 2;
My_Rec.Str          := 'Castro Street!';
My_String.LENGTH    := SIZE( My_Rec.Int ) + 2 + LENGTH( My_Rec.Str );
My_String.BODY::rt  := My_Rec;
WRITE( ISAM_File, My_String );

My_Rec.Int          := 1;
My_Rec.Str          := 'Harvey Milk welcomes you to';
My_String.LENGTH    := SIZE( My_Rec.Int ) + 2 + LENGTH( My_Rec.Str );
My_String.BODY::rt  := My_Rec;
WRITE( ISAM_File, My_String );

CLOSE( ISAM_File );
END.
```

Example 5–5 shows how to read from the ISAM file with varying-length records.

Example 5–5: Reading from an ISAM File with Varying-Length Records

```
{
Source File: READ_V_ISAM.PAS
This example reads from an ISAM file with varying-length records.
}
PROGRAM Idx_Read( OUTPUT );
TYPE
   rt = RECORD
      Int : INTEGER;
      Str : VARYING[132] OF CHAR;
      END;
   vt = VARYING[ SIZE( rt ) ] OF CHAR;
VAR
   My_Rec    : rt;
   ISAM_File : FILE OF vt;

BEGIN
OPEN( ISAM_File, 'v.idx', HISTORY := OLD, ACCESS_METHOD := KEYED,
      RECORD_TYPE := VARIABLE, ORGANIZATION := INDEXED );
RESET( ISAM_File );
```

(continued on next page)

Example 5–5 (Cont.): Reading from an ISAM File with Varying-Length Records

```
FINDK( ISAM_File, 0, 1);
My_Rec := ISAM_File^.BODY::rt;
WRITELN( My_Rec.Str );

FINDK( ISAM_FILE, 0, 2 );
My_Rec := ISAM_File^.BODY::rt;
WRITELN( My_Rec.Str );

CLOSE( ISAM_File );
END.
```

The following is output when you use the code in Examples 5–3, 5–4, and 5–5:

```
$  CREATE/FDL=MAKE_V_ISAM.FDL
$  RUN  WRITE_V_ISAM
$  RUN  READ_V_ISAM
Harvey Milk welcomes you to
Castro Street!
```

5.5 RMS and RTL Examples

Example 5–6 uses a user-action routine to initialize the I/O time-out period for a terminal.

Example 5–6: Using STATUS and PAS$RAB to Obtain the Status of I/O Operations

```
{
Source File: STATUS_PAS$FAB
This example shows how to use STATUS and PAS$RAB to obtain the status
of I/O operations.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Timeout( INPUT, OUTPUT );
CONST
   {
Including PASSTATUS.PAS defines the VAX Pascal error codes detected by
the STATUS and STATUSV functions.
   }
   %INCLUDE 'SYS$LIBRARY:PASSTATUS.PAS/NOLIST'

VAR
   Term_File : TEXT;
   Number    : INTEGER;
   Prompt    : PACKED ARRAY[1..33] OF CHAR
               VALUE 'Enter an integer (zero to quit): ';
{
Three parameters are passed to a user action routine:  the FAB, the
RAB, and the file variable.  Recall that FAB$TYPE and RAB$TYPE are
defined in STARLET.PEN.
}
FUNCTION Set_Timeout( VAR File_Fab : FAB$TYPE;
                      VAR File_Rab : RAB$TYPE;
                      VAR A_File   : TEXT ) : INTEGER;

   VAR
      Open_Status: INTEGER;
   BEGIN
   {
The file is opened with  an input time-out period of 10 seconds.
   }
   File_Rab.rab$b_tmo := 10;
   File_Rab.rab$v_tmo := TRUE;

   {
When a TEXT file is opened with a user action routine, the usual VAX
Pascal prompting feature is not enabled.   For instance, the following
two statements do not display the prompt:
               WRITE (Prompt);
               READLN (Term_File, Number);
Instead, RMS prompting is enabled for the file by setting the
appropriate field in the RAB.  Now any READ or GET operation on the
file translates to a "read with prompt."
   }
   File_Rab.rab$v_pmt := TRUE;
   File_Rab.rab$l_pbf := IADDRESS( Prompt );
   File_Rab.rab$b_psz := SIZE( Prompt );
```

(continued on next page)

Example 5–6 (Cont.): Using STATUS and PAS$RAB to Obtain the Status of I/O Operations

```
   {Open the file and connect the record stream:}
   Open_Status := $CREATE( Fab := File_Fab );
   IF ODD( Open_Status ) THEN $CONNECT( Rab := File_Rab );
   Set_Timeout := Open_Status
   END;       {Set_Timeout}

PROCEDURE Process_Get_Error;
   {
Determines the RMS error that caused a PAS$K_ERRDURGET
Global variable:  term_file
   }
   TYPE
      Rab_Ptr = ^RAB$TYPE;
   VAR
      Rab_Start : Rab_Ptr;

   FUNCTION PAS$RAB( VAR F : [UNSAFE] TEXT ) : Rab_Ptr; EXTERNAL;

   BEGIN  {Process_Get_Error}
   Rab_Start := PAS$RAB( Term_File );
   {
VAX Pascal only informs the program that an error occurred during the
READ (or GET).  To determine the actual RMS error code, procedure
Process_Get_Error calls PAS$RAB to obtain a pointer to the file $RAB.
The RAB$L_STS filed in the RAB contains the actual RMS status code.

If the RMS status code is RMS$TMO (timeout), the program informs the
user. Any other error is unexpected, so the program signals for a
condition handler.
   }
   IF Rab_Start^.rab$l_sts = rms$_tmo THEN
      WRITELN( 'No response in 10 seconds...' )
   ELSE LIB$SIGNAL( Rab_Start^.rab$l_sts,
                    Rab_Start^.rab$l_stv )
   END;   {Procedure}

BEGIN  {Main program}
OPEN( File_Variable := Term_File, File_Name := 'TT:',
      History:= NEW, User_Action:= Set_Timeout );
RESET( Term_File );
```

(continued on next page)

Example 5–6 (Cont.): Using STATUS and PAS$RAB to Obtain the Status of I/O Operations

```
{Accept input until zero entered:}
Number:= 1;
WHILE ( Number <> 0 ) DO
   BEGIN
   {
The program accesses the file using a normal READ statement.  When the
ERROR parameter is used, it must be specified using keyword calling
syntax. In addition, it must be the last parameter in the list.
   }
   READ( Term_File, Number, ERROR:= CONTINUE );
   {
The STATUS function returns the status code of the last operation on
the file.  If the status code is PAS$K_ERRDURGET, procedure
Process_Get_Error is called.
   }
   CASE STATUS( Term_File ) OF
      pas$k_success    : WRITELN( 'You entered: ', Number );
      pas$k_errdurget : Process_Get_Error;
                              {Extended-digit notation:}
      OTHERWISE  LIB$STOP( 16#218004 +
                 ( ( STATUS( Term_File ) + 200 ) * 8 ) )
      END;  {Case}

   {Clear the rest of the line:}
   WHILE NOT EOLN( Term_File )  DO
      GET( Term_File );
   END;  {While not zero}
END.
```

The following is output when you run the program in Example 5–6:

```
$  RUN STATUS_PAS$FAB
Enter an integer (zero to quit): 10
You entered:        10
Enter an integer (zero to quit):
No response in 10 seconds...
Enter an integer (zero to quit): 0
You entered:          0
```

Example 5–7 shows how to access a relative file randomly.

Example 5–7: Using Random Access on a Relative File

```
{
Source File: STATUS_RELATIVE.PAS
This program shows random access of a file of relative organization.
}
[INHERIT( 'sys$library:pascal$lib_routines' )]
PROGRAM Relative( INPUT, OUTPUT );
CONST
   Prompt = 'Record number (CTRL/Z to quit): ';
   %INCLUDE 'SYS$LIBRARY:PASSTATUS.PAS'   {VAX Pascal codes for STATUS}
TYPE
   Char_Array = PACKED ARRAY[1..10] OF CHAR;
VAR
   Rel_File : FILE OF Char_Array;
   Rec_Num  : INTEGER;

BEGIN   {Main program}

{If REL.DAT exists and is not relative, this generates an error:}
OPEN( Rel_File, 'REL.DAT', HISTORY := OLD,
      ORGANIZATION := RELATIVE, ACCESS_METHOD := DIRECT );

{Get records by record number until EOF}
WRITE( Prompt );
WHILE NOT EOF( Input ) DO
   BEGIN
   {Get record:}
   READLN( Rec_Num );
   {
The call to the FIND routine positions the file at the component
specified in Rec_Num.  Error := CONTINUE is specified, because the
program attempts to handle errors from the FIND procedure.
   }
   FIND( Rel_File, Rec_Num, ERROR:= CONTINUE );

   {
The STATUS function is used to detect file access errors.  The program
performs certain operations if the status is SUCCESS or ERRDURFIN.  If
any other errors occur, LIB$STOP is called.
   }
   CASE STATUS( Rel_File ) OF
      PAS$K_SUCCESS   : IF NOT UFB( Rel_File ) THEN
                           WRITELN( Rel_File^ )
                        ELSE
                           WRITELN( 'Did not find record ', Rec_Num );

      PAS$K_ERRDURFIN : WRITELN( 'Error during FIND.' );

      OTHERWISE {Signal the error}
                        LIB$STOP( 16#218004 +
                        ( ( STATUS( Rel_File ) + 200 ) * 8 ) );
```

(continued on next page)

Example 5–7 (Cont.): Using Random Access on a Relative File

```
   WRITE( Prompt );
   END;  {While}
END.
{
Sample file REL.DAT:

michael
maryanne
brian
armisted
madrigal
}
```

The following is output when you run the program in Example 5–7:

```
$ RUN STATUS_RELATIVE
Record number (CTRL/Z to quit):  2
maryanne
Record number (CTRL/Z to quit):  1
michael
Record number (CTRL/Z to quit):  30
Did not find record          30
Record number (CTRL/Z to quit):  CTRL/Z
```

Example 5–8 shows how to access an indexed file randomly.

Example 5–8: Keyed and Sequential Access to Indexed Files

```
{
Source File:  KEYED_ACCESS.PAS
This program prompts  for a department number then prints the name of
each person in the department.
}
PROGRAM Indexed( INPUT, OUTPUT );
LABEL 100;
CONST
   Prompt = 'Enter department no. (119,210,220): ';
TYPE
   Employee_Rec = RECORD
      Id_Num :   PACKED ARRAY[1..5] OF CHAR;
      Name   :   PACKED ARRAY[1..6] OF CHAR;
      Dept   :   PACKED ARRAY[1..3] OF CHAR;
      Skill  :   PACKED ARRAY[1..2] OF CHAR;
      Salary :   PACKED ARRAY[1..4] OF CHAR;
      END;
VAR
   Employees : FILE OF Employee_Rec;
   Dept_Key  : PACKED ARRAY[1..3] OF CHAR;

BEGIN
{
To allow random keyed access to an indexed file, the access method
parameter to the OPEN procedure must be INDEXED.

The location of the keys in the records need not be specified since
this information is obtained by RMS from the file prologue.
}
OPEN( Employees, 'IDX.DAT',    History := OLD,
      Organization := INDEXED, Access_Method := KEYED );
```

(continued on next page)

Example 5–8 (Cont.): Keyed and Sequential Access to Indexed Files

```
WRITELN( 'Type CTRL/Z to quit' );   {Obtain department number}
WRITE( Prompt );
WHILE NOT EOF( INPUT ) DO
   BEGIN
   READLN( Dept_key );
   {
A Key_Number of zero indicates the primary key, and Key_Value
specifies the value of the key. It is not necessary to call RESETK,
because the file need not be in inspection mode before FINDK is
executed.
   }
   FINDK( Employees, Key_Number := 0, Key_Value := Dept_Key );
   {If valid dept., output members:}
   IF NOT UFB( Employees ) THEN  {Valid dept number}
      WHILE NOT EOF( Employees ) DO
         BEGIN
         IF Employees^.Dept <> Dept_Key THEN  GOTO 100;
         {
Sequential reads are performed to obtain the records with the
appropriate department number.  RMS does not return a different status
code when the key value changes.  Consequently, the program must check
the key value in the record to determine whether or not it has
changed.
         }
         WRITELN( Employees^.Name );
         GET( Employees )
         END   {WHILE NOT EOF}
   ELSE  {Not valid dept. number}
      WRITELN( 'Invalid department number.' );

100:     {Prompt for another dept. number}
   WRITE( Prompt );
   END;  {WHILE NOT EOF}
END.
{
Required Input file: IDX.DAT   (Not supplied)
}
```

The following is output when you run the program in Example 5–8:

```
$ RUN KEYED_ACCESS
Type CTRL/Z to quit
Enter department no. (119,210,220):  119
ANDEWF
DALLJE
FLANJE
FLINGA
GREEJW
JONEKB
MANKCA
MARSJJ
REDFBB
SCHAWE
WIENSH
Enter department no. (119,210,220):  3
Invalid department number.
Enter department no. (119,210,220):  CTRL/Z
```

Example 5–9 uses LIB$FIND_FILE and LIB$FIND_FILE_END to process a wildcard file name expression.

Example 5–9: Using LIB$FIND_FILE to Process a Wildcard File Name

```
{
Source File: LIB$FIND_FILE.PAS
This program shows how to process a wildcard file name using
LIB$FIND_FILE.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Find_File( INPUT, OUTPUT );
VAR
    File_Spec   : VARYING[132] OF CHAR;
    Result_Spec : VARYING[132] OF CHAR;
    Context     : UNSIGNED;
    Ret_Stat    : UNSIGNED;
```

(continued on next page)

Example 5–9 (Cont.): Using LIB$FIND_FILE to Process a Wildcard File Name

```
BEGIN
Context := 0;
WRITE( 'Enter filespec to parse: ' );
WHILE NOT EOF DO
   BEGIN
   {Read the filespec:}
   READLN( File_Spec );
   {Loop and parse the file spec:}
   REPEAT
      {
Call LIB$FIND_FILE to parse and to return a file name that satifies the
wildcard file specification.  %DESCR provides a CLASS_VS descriptor.
      }
      Ret_Stat := LIB$FIND_FILE( File_Spec, %DESCR Result_Spec,
                                 Context );
      {
Determine if LIB$FIND_FILE successfully returned a filename.
Termination because of no more files (RMS$_NMF) or no such file
(RMS$_FNF) will be ignored.
      }
      IF ( NOT ODD( Ret_Stat ) )  AND
         ( Ret_Stat <> RMS$_NMF ) AND
         ( Ret_Stat <> RMS$_FNF ) THEN  LIB$STOP( Ret_Stat );

      IF ( Ret_Stat <> RMS$_NMF ) AND
         ( Ret_Stat <> RMS$_FNF ) THEN
           WRITELN(Result_Spec);
   {
Continue calling LIB$FIND_FILE until there are no more files that
satisfy the wildcard file specification.
   }
   UNTIL ( Ret_Stat = RMS$_NMF ) OR ( Ret_Stat = RMS$_FNF );
   {
Call LIB$FIND_FILE_END to deallocate the context from the calls
to LIB$FIND_FILE so they won't affect the program when it parses
for the next wildcard file specification.
   }
   LIB$FIND_FILE_END( Context );

   {Get another file spec:}
   WRITE( 'Enter filespec to parse: ' );
   END;  {WHILE NOT EOF}
END.
```

The following is output when you run the program in Example 5–9:

```
Enter filespec to parse:  DATA*.DAT
EX$:[EXAMPLE]DATA.DAT;6
EX$:[EXAMPLE]DATA_FILE.DAT;2
EX$:[EXAMPLE]DATA_FILE2.DAT;1
Enter filespec to parse:  CTRL/Z
```

5.6 Condition Handler Examples

Example 5–10 shows the use of a condition handler.

Example 5–10: Using a Condition Handler

```
{
Source File: HANDLER.PAS
This program demonstrates how to write  a condition handler that
catches an error, sets an outer-level variable, and uses a GOTO to go
to a cleanup section.
}
[INHERIT( 'sys$library:starlet' )]
PROGRAM Condition_Handler( INPUT, OUTPUT );

   PROCEDURE Level1;
      LABEL  Error_Return;
      TYPE
        Signal_Array    = ARRAY[0..1] OF INTEGER;
        Mechanism_Array = ARRAY[0..4] OF INTEGER;
        Large_Basetype  = ARRAY[0..MAXINT DIV 16] OF CHAR;
      VAR
         Large_Array    : ^Large_Basetype;
         Handler_Status : [VOLATILE] INTEGER VALUE 0;

      [ASYNCHRONOUS] FUNCTION Handler(
            VAR Sig_Args  : Signal_Array;
            VAR Mech_Args : Mechanism_Array ) : INTEGER;
      BEGIN
      IF Sig_Args[1] <> ss$_unwind THEN
         BEGIN
         Handler_Status := Sig_Args[1];
         GOTO Error_Return;
         END
      ELSE
         Handler := ss$_resignal;
      END;  {Function Handler}
```

(continued on next page)

Example 5–10 (Cont.): Using a Condition Handler

```
   BEGIN {Procedure Level1}
   ESTABLISH( Handler );
   {
Try to allocate a very large array that results in an error being
signalled  by the call to NEW.
   }
   NEW( Large_Array );

   Error_Return:
   IF Handler_Status <> 0 THEN
      WRITELN( 'Handler_Status = ', HEX( Handler_Status ) );
   END;  {Procedure Level1}

BEGIN  {Program Condition_Handler}
Level1;
END.
```

5.7 System Services Examples

Example 5–11 shows how to use asynchronous system traps (ASTs).

Example 5–11: Using Asynchronous System Traps (ASTs)

```
{
Source File: AST.PAS
This program sets a 10 second timer which requests  an AST.  The main
program then performs arithmetic  operations for the user, interrupted
after 10 seconds  by the timer AST.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Timer_Ast( INPUT, OUTPUT );
CONST
   Delay = '0 ::10.00';
TYPE
   Quadword = RECORD
      Lo : UNSIGNED;
      Hi : INTEGER;
      END;
VAR  Bin_Delay             : Quadword;
     Ast_Output            : [VOLATILE] TEXT;
     Num1, Num2, Sys_Stat : INTEGER;
{
Ast_Proc must be declared UNBOUND, because it is passed to $SETIMR using
%IMMED.
}
[UNBOUND,ASYNCHRONOUS] PROCEDURE Ast_Proc;
   {
This routine is called as an AST procedure.  It prints the current
time at the terminal.
   }
   VAR
      Cur_Time  :  PACKED ARRAY[1..23]OF CHAR;
      Time_Stat :  INTEGER;
   BEGIN
      Time_Stat:= LIB$DATE_TIME( Cur_Time );
      IF NOT ODD( Time_Stat ) THEN LIB$STOP( Time_Stat );
      WRITELN( Ast_Output );
      {
When the AST is delivered, the AST routine interrupts the program and
outputs this message.
      }
      WRITELN( Ast_Output,'   The time is now:  ', Cur_Time );
      WRITELN( Ast_Output )
   END;  { Ast_Proc }
```

(continued on next page)

Example 5–11 (Cont.): Using Asynchronous System Traps (ASTs)

```
BEGIN   {Main program}
OPEN( Ast_Output, 'TT:' ); {Output channel for Ast_Proc}
{
The AST routine needs an output channel to the terminal, separate from
the channel used by the main program.  The channel is opened and
associated to the terminal (TT:) in the main program, because this is
a time-consuming operation.  You should avoid including unnecessary
lengthy operations in AST routines.

Any global variables, like AST_OUTPUT, referenced by a routine
declared ASYNCHRONOUS,  must be declared VOLATILE.  This attribute
informs the compiler that the variable is subject to change at any
time, and should be optimized carefully.
}

REWRITE( Ast_Output );

{Convert delay interval to binary format and set timer:}
Sys_Stat := $BINTIM( Delay, Bin_Delay );

IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
{
The ASTADR parameter in the call to $SETIMR is the address of
the entry point of an AST routine.  The AST routine will be
executed when the timer expires.  If the AST routine is to be
executed repeatedly, rather than just once, the timer should be
reset at the end of the AST routine.
}
Sys_Stat := $SETIMR( Daytim:= Bin_Delay, Astadr:= Ast_Proc );
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );

{Prompt user for two numbers and multiply them:}
REPEAT
   WRITELN( 'Enter two integers to be multiplied.' );
   WRITELN( '(Enter two zeros to quit.)' );
   READLN( Num1, Num2 );
   WRITELN( ' The product is: ', Num1 * Num2 )
UNTIL ( Num1 = 0 ) AND ( Num2 = 0 );
WRITELN( 'Arithmetic operations completed.' );
END.
```

The following is output when you run the program in Example 5–11:

```
$  RUN AST
Enter two integers to be multiplied.
(Enter two zeros to quit.)
12
12
The product is:         144
Enter two integers to be multiplied.
(Enter two zeros to quit.)
```

```
   The time is now: 11-OCT-1984  16:18:43.54
0
0
The product is:           0
Arithmetic operations completed.
```

Example 5–12 performs output to a terminal via the $QIOW system service.

Example 5–12: Using the SYS$QIOW Routine

```
{
Source File: SYS$QIOW.PAS
This program shows the use of the $QIOW system  service to
perform synchronous I/O to a terminal.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Qiow( OUTPUT );
CONST
   Text_String = 'This is from a $QIOW.';
   Terminal    = 'TT:';
TYPE
   Word_Integer = [WORD] 0..65535;
   Io_Block = RECORD
      Io_Stat, Count : Word_Integer;
      Dev_Info       : INTEGER
      END;
VAR
   Term_chan    : Word_Integer;
   Counter      : INTEGER;
   Sys_Stat     : INTEGER;
   Iostat_Block : Io_Block;

BEGIN
{
TERM_CHAN receives the channel number from the $ASSIGN system
service.  The process permanent logical name SYS$OUTPUT is
assigned to your terminal when you login. The $ASSIGN system
service will translate the logical name to the actual device name.
}
Sys_Stat := $ASSIGN( Terminal, Term_Chan , , );
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );

{Output the message twice:}
FOR Counter := 1 TO 2 DO
   BEGIN
   {
The function IO$_WRITEVBLK requires values for parameters P1, P2, and P4.
P1 is the starting address of the buffer containing the message.  P2 is the
number of bytes to be written to the terminal (in this example, 21).  P4 is
the carriage control specifier; 32 indicates single space carriage control.
```

(continued on next page)

Example 5–12 (Cont.): Using the SYS$QIOW Routine

```
$QIOW is used, instead of $QIO, to insure that the output operation will
complete before the program terminates.
   }
   Sys_Stat := $QIOW( Chan := Term_Chan,
                      Func := IO$_WRITEVBLK,
                      Iosb := Iostat_Block,
                      P1   := Text_String,
                      P2   := LENGTH( Text_String ),
                      P4   := 32 );
   IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
   IF NOT ODD( Iostat_Block.Io_Stat ) THEN
      LIB$STOP( Iostat_Block.Io_Stat );
   END;  {For Counter}
END.
```

Example 5–13 shows how to create a global section and use it to transfer data between two processes. This example requires that you run programs GLOBAL1 and GLOBAL2 in the same UIC group. GLOBAL1 creates and maps the section, but GLOBAL2 only maps the section. Therefore, GLOBAL1 must be run first.

Example 5–13: Mapping Global Sections (GLOBAL1.PAS)

```
{
Source File: GLOBAL1.PAS
This program creates and maps a global section.  Data in the section
file is accessed through an  array.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Global1( OUTPUT );
TYPE
   Int_Pointer  = ^INTEGER;          {For addresses}
   File_Type    = FILE OF INTEGER;
   Word_Integer = [WORD] 0..65535;
VAR
   {
The $CRMPSC system service maps pages starting at page boundaries.
The ALIGN attribute, with a value of 9, is used to ensure that IARRAY
starts on a page boundary.  (If the binary address of the first
element of IARRAY ends in 9 zeros, then it is page aligned.)
   }
   My_Adr, Sys_Adr : ARRAY[1..2] OF Int_Pointer;
   Iarray : [VOLATILE, ALIGNED( 9 ),BYTE( 512 )]
                        ARRAY[1..50] OF INTEGER;
   Sec_Chan           :  UNSIGNED;
   Name               :  PACKED ARRAY[1..4] OF CHAR;
   Sec_File           :  File_Type;
   Sec_Flags          :  INTEGER;
   Sys_Stat, Counter  :  INTEGER;

FUNCTION Get_Chan( VAR File_Fab : FAB$TYPE; VAR File_Rab : RAB$TYPE;
                   VAR A_File   : File_Type ) : INTEGER;
   {
This routine is called as a user_action function by the OPEN
statement.  The channel number for file is stored in the global
variable SEC_CHAN.
   }
   VAR
      Open_Status : INTEGER;

   BEGIN
   {Define appropriate FAB fields:}
   File_Fab.fab$v_cbt := FALSE;
   File_Fab.fab$v_ctg := TRUE;
   File_Fab.fab$v_ufo := TRUE;
   File_Fab.fab$l_alq := 1;
```

(continued on next page)

Example 5–13 (Cont.): Mapping Global Sections (GLOBAL1.PAS)

```
   {Open the file:}
   Open_Status := $CREATE( FAB := File_Fab );
   IF ODD( Open_Status ) THEN
      BEGIN
      {
Because the user-file-open option (FAB$V_UFO) was specified in the
file FAB, the file channel number is stored in the file FAB at offset
FAB$L_STV.
      }
      Sec_Chan:= File_Fab.fab$l_stv
      END;
   Get_Chan:= Open_Status
   END;   {Get_Chan}

BEGIN  {Main program}

{Associate with common cluster MYCLUS:}
$ASCEFC( Efn := 64, Name:= 'MYCLUS' );
{
In order to map a disk file as a section, we need a channel for the
file. A channel is assigned by the OPEN statement, and the user-action
feature of the OPEN statement allows you to obtain the number of the
assigned channel.
}
OPEN( File_Variable := Sec_File, File_Name := 'section.dat',
      History := New, User_Action := Get_Chan );

{Obtain starting and ending addresses of the section:}
My_Adr[1] := ADDRESS( Iarray[1] );
My_Adr[2] := ADDRESS( Iarray[50] );
{
The starting and ending process virtual addresses of the section are
placed in MY_ADR.  The output argument SYS_ADR receives the starting
and ending system virtual addresses.  The flag SEC$M_GLOBAL requests a
global section.  The flag SEC$M_WRT indicates that the pages should be
writeable as well as readable.  The SEC$M_DZRO flag requests pages
filled with zeros.
}
Name      := 'GSEC';
Sec_Flags := sec$m_gbl + sec$m_wrt + sec$m_dzro;
Sys_Stat  := $CRMPSC( My_Adr, Sys_Adr,, Sec_flags, Name,,,
                      %IMMED Sec_Chan, 1,,, );

IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
```

(continued on next page)

Example 5–13 (Cont.): Mapping Global Sections (GLOBAL1.PAS)

```
{Write data to the global section:}
FOR Counter := 1 TO 50 DO
   Iarray[Counter] := Counter;
{
Data is placed in the global section and then the section is updated
with the $UPDSEC system service.  The $UPDSEC system service executes
asynchronously.  Therefore, event flag 1 is used to synchronize
completion of the update operation.
}
Sys_Stat := $UPDSEC( Inadr := Sys_Adr, Efn := 1 );
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );

{Wait until pages have been written back to the disk:}
Sys_Stat := $WAITFR( Efn := 1 );
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );

{Wait for GLOBAL2 to update the section:}
Sys_Stat := $SETEF( Efn := 72 );
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
WRITELN( 'Waiting for GLOBAL2 to update section.' );
Sys_Stat := $WAITFR( Efn:= 73 );
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );

{Print the modified pages:}
WRITELN( 'Modified data in the global section:' );
FOR Counter := 1 TO 50 DO
   BEGIN
   WRITE( Iarray[Counter]:5 );
   IF ( Counter REM 10 ) = 0 THEN WRITELN;
   END;
END.
```

Example 5–14 shows the second program to be run with the program in Example 5–13.

Example 5–14: Mapping Global Sections (GLOBAL2.PAS)

```
{
Source File: GLOBAL2.PAS
This program maps and modifies a global section after GLOBAL1
creates it.  Programs GLOBAL1 and  GLOBAL2 synchronize the
processing of the global  section through common event flags.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Global2( OUTPUT );
TYPE
   Int_Pointer = ^INTEGER;           {For addresses}
   File_Type   = FILE OF INTEGER;
VAR
   My_Adr : ARRAY[1..2] OF Int_Pointer;
   Iarray : [VOLATILE, ALIGNED( 9 ), BYTE( 512 )]
            ARRAY[1..50] OF INTEGER;
   Sys_Stat, Counter : INTEGER;

BEGIN
{Obtain starting and ending addresses of the section:}
My_Adr[1] := ADDRESS( Iarray[1] );
My_Adr[2] := ADDRESS( Iarray[50] );
{
Associate with common cluster MYCLUS, and wait for
event flag to be set.
}
Sys_Stat := $ASCEFC( Efn := 64, Name := 'MYCLUS' );
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
Sys_Stat := $WAITFR( Efn:= 72 );

IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
{
GLOBAL2 maps the existing section as writeable by specifying the
SEC$M_WRT flag.  Note that the SEC$M_DZRO flag is not set, because
that would destroy any data that might be in the section.
}
Sys_Stat := $MGBLSC( Inadr  := My_Adr, Flags := SEC$M_WRT,
                     Gsdnam := 'GSEC' );

IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );

{Output data in global section, and double each value:}
WRITELN( 'Original data in global section:' );
```

(continued on next page)

Example 5–14 (Cont.): Mapping Global Sections (GLOBAL2.PAS)

```
FOR Counter := 1 TO 50 DO
   BEGIN
   WRITE( Iarray[Counter]:5 );
   IF ( Counter REM 10 ) = 0 THEN WRITELN;
      Iarray[Counter] := Iarray[Counter] * 2
   END;

{Set event flag to allow GLOBAL1 to continue:}
Sys_Stat:= $SETEF( Efn := 73 );
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
END.
```

The following is output when you run the program in Example 5–13 first and then run the program in Example 5–14 from another process:

```
$ SPAWN
%DCL-S-SPAWNED, process EXAMPLE_1 spawned
%DCL-S-ATTACHED, terminal now attached to process EXAMPLE_1
$ ATTACH EXAMPLE
%DCL-S-RETURNED, control returned to process EXAMPLE
$ RUN  GLOBAL1
Waiting for GLOBAL2 to update section.
CTRL/Y
Interrupt
$ ATTACH EXAMPLE_1
%DCL-S-RETURNED, control returned to process EXAMPLE_1
$ RUN  GLOBAL2
Original data in global section:
    1    2    3    4    5    6    7    8    9   10
   11   12   13   14   15   16   17   18   19   20
   21   22   23   24   25   26   27   28   29   30
   31   32   33   34   35   36   37   38   39   40
   41   42   43   44   45   46   47   48   49   50
$ LOGOUT
  Process EXAMPLE_1 logged out at 25-SEP-1989 11:17:59.42
$ CONTINUE
Modified data in the global section:
    2    4    6    8   10   12   14   16   18   20
   22   24   26   28   30   32   34   36   38   40
   42   44   46   48   50   52   54   56   58   60
   62   64   66   68   70   72   74   76   78   80
   82   84   86   88   90   92   94   96   98  100
```

Example 5–15 shows a program that is used to update records in a relative file. The OPEN option SHARING := READWRITE is specified to invoke the RMS file sharing facility. In this example, the same program is used to access the file from two processes.

Example 5–15: File Sharing from Several Processes

```
{
Source File: FILE_SHARING.PAS
This program, when run from several processes, shares access to a
relative file called REL.DAT.
}
[INHERIT( 'sys$library:pascal$lib_routines' )]
PROGRAM Sharedfil( INPUT, OUTPUT );
CONST
   Prompt = 'Record number (CTRL/Z to quit): ';
   {
The PASSTATUS.PAS file is included to define the symbolic status codes
for the conditions detected by the STATUS function.  No semicolon is
necessary to end the INCLUDE clause.
   }
   %INCLUDE 'SYS$LIBRARY:PASSTATUS.PAS'

TYPE
   Char_Array = PACKED ARRAY[1..10] OF CHAR;
VAR
   Rel_File : FILE OF Char_Array;
   Rec_Num  : INTEGER;

PROCEDURE Process_Rec;
   {Output a record and modify value, if necessary:}
   BEGIN
      WRITELN( 'file record is: ', Rel_File^ );
      WRITE( 'New Value or CR: ' ); {Request updated record}
      IF NOT EOLN( INPUT ) THEN
         BEGIN
         {
Read new information from SYS$INPUT directly into the file buffer
         }
         READLN( Rel_File^ );
         UPDATE( Rel_File );
         IF STATUS( Rel_File ) <> 0 THEN
            LIB$STOP( 16#218004 +
            ( ( STATUS( Rel_File ) + 200 ) * 8 ) )
         END   {IF NOT EOLN}
         ELSE READLN  {Read past the EOLN}
   END;  {Process_Rec }
```

(continued on next page)

Example 5–15 (Cont.): File Sharing from Several Processes

```
BEGIN   {Main program}
{
SHARING := READWRITE is specified to OPEN to indicate that the file
can be shared.  Because manual locking was not specified, RMS
automatically controls access to the file.
}
OPEN( FILE_VARIABLE := Rel_File, FILE_NAME := 'REL.DAT',
      HISTORY := Old,            ACCESS_METHOD := Direct,
      ORGANIZATION := Relative,  SHARING := Readwrite );
WRITE( Prompt );
WHILE NOT EOF( Input ) DO
   BEGIN
   READLN( Rec_Num );  {Get record}
   FIND( Rel_File, Rec_Num, ERROR:= CONTINUE );
   CASE STATUS( Rel_File ) OF
      pas$k_success : IF NOT UFB( Rel_File ) THEN
                         Process_Rec
                      ELSE WRITELN ( 'Did not find record ',
                                     Rec_Num );
      pas$k_faigetloc : WRITELN( 'Record locked.' );
      pas$k_errdurfin : WRITELN( 'Error during FIND.' );
      OTHERWISE  { signal the error }
            LIB$STOP( 16#218004 +
                    ( ( STATUS( Rel_File ) + 200 ) * 8 ) );
      END;   {CASE}
   WRITE( Prompt );
   END;  {WHILE}
END.
{
Sample file REL.DAT:

michael
maryanne
brian
armisted
madrigal
}
```

The following is output when you run the program in Example 5–15:

```
$ SPAWN
%DCL-S-SPAWNED, process EXAMPLE_1 spawned
%DCL-S-ATTACHED, terminal now attached to process EXAMPLE_1
$ ATTACH EXAMPLE
%DCL-S-RETURNED, control returned to process EXAMPLE
$ RUN SHAREDFIL
Record number (CTRL/Z to quit):  2
file record is:  maryanne
New Value or CR:  CTRL/Y
Interrupt
$ ATTACH EXAMPLE_1
%DCL-S-RETURNED, control returned to process EXAMPLE_1
$ RUN SHAREDFIL
Record number (CTRL/Z to quit):  2
Record locked.
Record number (CTRL/Z to quit):  3
file record is:  brian
New Value or CR:  CTRL/Y
Interrupt
$ ATTACH EXAMPLE
%DCL-S-RETURNED, control returned to process EXAMPLE
$ CONTINUE
Record number (CTRL/Z to quit):  3
Record locked.
Record number (CTRL/Z to quit):  CTRL/Z
ATTACH EXAMPLE_1
%DCL-S-RETURNED, control returned to process EXAMPLE_1
$ CONTINUE
CTRL/Z
$ LOGOUT
  Process EXAMPLE_1 logged out at 25-SEP-1989 11:17:59.42
```

Example 5–16 shows the use of a condition handler and the SYS$DCLEXH routine.

Example 5–16: SYS$DCLEXH and Condition Handlers

```
{
Source File: SYS$DCLEXH.PAS
This program declares an exit handler.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' ),
IDENT( 'V1.0' )]    PROGRAM Exit_Hand( OUTPUT );
TYPE
   Exit_Control_Block = RECORD
      Forward_Link     : INTEGER;
      Handler_Address  : INTEGER;
      Parameter_Count  : INTEGER;
      First_Parameter  : INTEGER;
      Second_Parameter : INTEGER;
      END;
VAR
   Reason, Second, Return_Status : [VOLATILE] INTEGER;
   p                             : ^INTEGER;
   Control_Block : Exit_Control_Block :=
      ( 0,  { Filled in by system }
        0,  { Filled at run time  }
        2,  { We have two parms   }
        0,  { Filled at run time  }
        0 );{ Filled at run time  }

PROCEDURE Exit_Handler( Reason, Second : INTEGER );
   BEGIN
   WRITELN( 'The reason is ', HEX( Reason ) );
   WRITELN( 'The second is ', Second );
   END;

BEGIN
{Fill in any fields in control_block:}
WITH Control_Block DO
   BEGIN
   Handler_Address  := IADDRESS( Exit_Handler );
   First_Parameter  := IADDRESS( Reason );
   Second_Parameter := IADDRESS( Second );
   END;

{Fill second parameter that will be printed by exit_handler:}
Second := 1010;

Return_Status := $DCLEXH( Control_Block );
IF NOT ODD( Return_Status ) THEN  LIB$STOP( Return_Status );

{Now finish by causing an ACCVIO:}
p := NIL;
p^ := 5;
END.
```

Example 5–17 shows how to use SMG$ screen management routines from VAX Pascal.

Example 5–17: Using SMG Routines

```
{
Source File: SGM_EXAMPLE.PAS
This program clears the screen, then divides it into two regions.  In
the top half of the screen, data is entered and echoed; in the bottom
half of the screen, the data is echoed again.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$smg_routines',
'sys$library:pascal$lib_routines' )]
PROGRAM SMG_Example;
TYPE
   Cond_Code          = UNSIGNED;
   Mask               = UNSIGNED;
   Display_ID_Type    = UNSIGNED;
   Pasteboard_ID_Type = UNSIGNED;
   Keyboard_ID_Type   = UNSIGNED;
   Key_Table_ID_Type  = UNSIGNED;
   Unsigned_Byte      = [BYTE] 0..255;
   Unsigned_Word      = [WORD] 0..65535;
VAR
   Stat          : Cond_Code;
   Text          : VARYING[132] OF CHAR;
   Keyboard_ID   : Keyboard_ID_Type;
   Pasteboard_ID : Pasteboard_ID_Type;
   Key_Table_ID  : Key_Table_ID_Type;
   Display_ID_1  : Display_ID_Type;
   Display_ID_2  : Display_ID_Type;
   PB_Rows       : INTEGER;
   PB_Columns    : INTEGER;
   Display_Rows  : INTEGER;

BEGIN
{
The CREATE_PASTEBOARD routine erases the screen by default.
It also returns the size of the output device which is used
to create the virtual displays.
}
Stat := SMG$CREATE_PASTEBOARD( Pasteboard_ID,
        Number_Of_Pasteboard_Rows    := PB_Rows,
        Number_Of_Pasteboard_Columns := PB_Columns );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

Stat := SMG$CREATE_VIRTUAL_KEYBOARD( Keyboard_ID );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

Stat := SMG$CREATE_KEY_TABLE( Key_Table_ID );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );
```

(continued on next page)

Example 5–17 (Cont.): Using SMG Routines

```
{
Create and paste two virtual displays: one for top half of screen and
another for bottom half of screen.  Label the bottom display with
"ECHO."
}
Display_Rows := ( PB_Rows - 4 ) DIV 2;

Stat := SMG$CREATE_VIRTUAL_DISPLAY( Display_Rows,
        PB_Columns, Display_ID_1 );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

Stat := SMG$CREATE_VIRTUAL_DISPLAY( Display_rows,
        PB_Columns, Display_ID_2 );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

Stat := SMG$PASTE_VIRTUAL_DISPLAY( Display_ID_1, Pasteboard_ID,
        1, 1 );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

Stat := SMG$PASTE_VIRTUAL_DISPLAY( Display_ID_2, Pasteboard_ID,
        PB_Rows DIV 2, 1 );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

Stat := SMG$LABEL_BORDER( Display_ID_2, 'ECHO' );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

{
This loop, consisting of calls to READ_COMPOSED_LINE and PUT_LINE,
causes the data to be echoed in both parts of the screen.
READ_COMPOSED_LINE transparently implements multi-line recall similar
to the multi-line command recall available in DCL.
}
WHILE ODD( SMG$READ_COMPOSED_LINE( Keyboard_ID, Key_table_ID,
           Text.BODY, '> ', Text.LENGTH, Display_ID_1 ) ) DO
   BEGIN
   Stat := SMG$PUT_LINE( Display_ID_2, Text );
   IF NOT ODD( Stat ) THEN LIB$STOP( Stat );
   END;

{
The call to SET_PHYSICAL_CURSOR leaves the cursor positioned
at the bottom of the screen when the program exits.
}
SMG$SET_PHYSICAL_CURSOR( Pasteboard_ID, PB_rows - 1, 1 );

END.
```

Example 5–18 shows how to use $QIOs.

Example 5–18: Using SYS$QIO Routines

```
{
Source File: SYS$QIO.PAS
This program shows the use of $QIOs to the  terminal and establishing
an AST routine to handle CTRL/C's.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Ctrl_C( INPUT, OUTPUT );
TYPE
   Word_Integer = [WORD] 0..65535;
   Io_Status = RECORD
      Io_Stat, Count : Word_Integer;
      Device_Info    : INTEGER
      END;
VAR
     Prompt       : PACKED ARRAY[1..48] OF CHAR;
     Buffer       : VARYING[80] OF CHAR;
     TT_Chan      : [VOLATILE] Word_Integer;
     Ast_Output   : [VOLATILE] TEXT;
     Io_Func      : Word_Integer;
     Iostat_Block : [VOLATILE] Io_Status;
     Counter      : INTEGER;
     Sys_Stat     : INTEGER;
{
Because CAST is called asynchronously:

  o  The procedure must be declared ASYNCHRONOUS and UNBOUND.
  o  LIB$STOP, which is called from the procedure, must also be declared
     ASYNCHRONOUS.
  o  TT_Chan must be declared VOLATILE.
  o  A separate output channel to the terminal (AST_Output) must be
     established.
}

{This procedure is called as a CTRL/C ast routine:}
[ASYNCHRONOUS, UNBOUND] PROCEDURE Cast( Channel : Word_Integer );
   VAR
      Cancel_Stat :  INTEGER;
   BEGIN
   {Cancel all I/O on terminal channel:}
   Cancel_Stat := $CANCEL( Channel );
   IF NOT ODD( Cancel_Stat ) THEN LIB$STOP( Cancel_Stat );

   { Tell user that CTRL/C AST routine was entered:}
   WRITELN( Ast_Output, 'You typed a CTRL/C' )
   END;    {PROCEDURE Cast}
```

(continued on next page)

Example 5–18 (Cont.): Using SYS$QIO Routines

```
BEGIN    {Begin main program}
OPEN( Ast_Output, 'TT:');     {Output channel for cast}
REWRITE( Ast_Output );
Prompt :='You have 5 seconds to enter data, or type CTRL/C';

{Assign channel to terminal:}
Sys_Stat := $ASSIGN( Devnam := 'SYS$COMMAND', Chan := TT_Chan );

IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
{
A CTRL/C AST routine is declared:

    o Note that this routine will only be entered for the first CTRL/C the
      user types.
    o The formal parameter for P1 specifies that it will be passed %REF.  In
      this case, you are passing the entry point for a routine, which
      should be passed %IMMED.  The foreign mechanism specifier is
      used on the actual parameter for P1 to override the formal definition.
    o The %REF mechanism specifier is used on the actual parameter for P2 to
      override the formal definition of %IMMED.  The CAST procedure
      expects this parameter to be passed to it by reference.
}
Io_Func  := io$_setmode + io$m_ctrlcast;
Sys_Stat := $QIOW( Chan := TT_Chan,     Func := Io_Func,
                   Iosb := Iostat_Block,
                   P1   := %IMMED Cast, P2 := %REF TT_Chan );
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
IF NOT ODD( Iostat_Block.Io_Stat ) THEN
   LIB$STOP( Iostat_Block.Io_Stat );
{
This request prompts a user for input. It sets up a time limit for
waiting between characters typed (five seconds). Also, any characters
typed will not be echoed on the terminal. An I/O status block is
established to contain the final I/O status and byte count for
characters read.

The %REF specifier is used on the actual P5 parameter to override type
checking against the formal parameter definition, and to pass the
parameter by reference rather than %IMMED.
}
Io_Func  := io$_readprompt + io$m_noecho + io$m_timed;

{
The LENGTH field of the varying string BUFFER is filled in manually
because it is not done by the system service.  System services can
accept (and manipulate) only fixed-length strings.
```

(continued on next page)

Example 5–18 (Cont.): Using SYS$QIO Routines

```
BUFFER could be defined as a fixed-length string, but you would have
to know (at compile-time) the exact size of the string to be read by
the $QIOW.
}
Sys_Stat := $QIOW( , TT_Chan, Io_Func, Iostat_Block,,,
                     Buffer.BODY, SIZE( Buffer.BODY ), 5,,
                     %REF Prompt, SIZE( Prompt ) );
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
Buffer.LENGTH := Iostat_Block.Count;

{Check status of I/O and issue appropriate message}
IF Iostat_Block.Io_Stat = ss$_normal THEN
   WRITELN( 'You typed: ', Buffer )
ELSE
   IF Iostat_Block.Io_Stat = ss$_controlc THEN
      {
Specific tests are made for CTRL/C or timeout detection using the
final I/O status in the I/O Status Block.
      }
      WRITELN( 'Request was canceled' )
   ELSE
      IF Iostat_Block.Io_Stat = SS$_TIMEOUT THEN
         WRITELN( 'You did not respond in time' )
      ELSE WRITELN( 'Unexpected status: ',
                    Iostat_Block.Io_Stat );
END.
```

The following is output when you run the program in Example 5–18:

```
$  RUN SYS$QIO
You have 5 seconds to enter data, or type CTRL/C
You typed: Christopher street
$  RUN EXAMPLE
You have 5 seconds to enter data, or type CTRL/C
CTRL/C

You typed a CTRL/C
Request was canceled
$  RUN EXAMPLE
You have 5 seconds to enter data, or type CTRL/C
You did not respond in time
```

Example 5–19 shows how to adjust the size of the process working set from a program.

Example 5–19: Adjusting the Working Set Size Using SYS$ADJWSL

```
{
Source File: WORKING_SET.PAS
This program shows how to adjust the size of the process working set
from a program.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Adjust( OUTPUT );
CONST
   Index = 10;
VAR
   Adjust_Amt :  INTEGER;
   New_Limit  :  UNSIGNED;
   Sys_Stat   :  INTEGER;

BEGIN
Adjust_Amt := -50;
WHILE Adjust_Amt <= 70 DO
   BEGIN
   {
The $ADJWSL is used to increase or decrease the number of pages in the
process working set.
   }
   Sys_Stat := $ADJWSL( Adjust_Amt, New_Limit );
   IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
   WRITELN( 'Modify working set by ', Adjust_Amt,
            '  New working set size = ', New_Limit );
   Adjust_Amt:= Adjust_Amt + 10;
   END;
END.
```

The following is output when you run the program in Example 5–19:

```
$  SHOW  WORKING_SET
  Working Set  /Limit=  150  /Quota=  200  /Extent=  200
  Adjustment enabled   Authorized Quota=  200
  Authorized Extent= 300
$  RUN WORKING_SET
Modify working set by -50    New working set size = 100
Modify working set by -40    New working set size = 60
Modify working set by -30    New working set size = 40
Modify working set by -20    New working set size = 40
Modify working set by -10    New working set size = 40
Modify working set by   0    New working set size = 40
Modify working set by  10    New working set size = 50
Modify working set by  20    New working set size = 70
Modify working set by  30    New working set size = 100
Modify working set by  40    New working set size = 140
Modify working set by  50    New working set size = 190
Modify working set by  60    New working set size = 200
Modify working set by  70    New working set size = 200
```

Notice that the program cannot decrease the working set limit beneath the minimum established by the operating system. Similarly, the process working set cannot be expanded beyond the authorized quota.

Example 5–20 shows how to obtain text from a HELP library. After the initial help request has been satisfied, the user is prompted and can request additional information.

Example 5–20: Accessing a System HELP Library

```
{
Source File: GET_HELP.PAS
This program looks up HELP text in SYS$HELPLIB.HLB and displays
it.
}
[INHERIT( 'sys$library:pascal$lib_routines' )]
PROGRAM Gethelp( INPUT, OUTPUT );
CONST
   {
The value associated with this particular code is obtained from the
$LBRDEF macro in STARLET.MLB.  The code is used to specify the
operation that is to be performed on a library.
   }
   LBR$C_READ = 16#01;

TYPE
   Word_Integer = [WORD] 1..65535;
   Char_String = VARYING[32] OF CHAR;
   Int_Array = ARRAY[1..3] OF INTEGER;
VAR
   Key     : ARRAY[1..3] OF Char_String;
   Key_Len : ARRAY[1..3] OF INTEGER;
   Lib_Index, Help_Stat, Counter : INTEGER;

FUNCTION LBR$INI_CONTROL( VAR Library_Index : INTEGER;
   Func : INTEGER; Libe_Type : INTEGER := %IMMED 0;
   VAR Namblk : ARRAY[ 1..U : INTEGER ]
                OF INTEGER := %IMMED 0 ) : INTEGER; EXTERNAL;

FUNCTION LBR$OPEN( Library_Index : INTEGER;
   Fns : [CLASS_S] PACKED ARRAY[ 1..U : INTEGER ] OF CHAR := %IMMED 0;
   Create_Options : Int_Array := %IMMED 0;
   Dns : [CLASS_S] PACKED ARRAY[ 12..u2 : INTEGER ] OF CHAR := %IMMED 0;
   Rlfna : ARRAY[ 13..u3 : INTEGER ] OF INTEGER := %IMMED 0;
   Rns : [CLASS_S] PACKED ARRAY[ 14..u4:INTEGER ] OF CHAR := %IMMED 0;
   VAR Rnslen : INTEGER := %IMMED 0 ) : INTEGER; EXTERNAL;
```

(continued on next page)

Example 5–20 (Cont.): Accessing a System HELP Library

```
FUNCTION LBR$GET_HELP( Library_Index : INTEGER;
   Line_Width : INTEGER := %IMMED 0;
   %IMMED [UNBOUND] PROCEDURE Routine := %IMMED 0;
   data : INTEGER := %IMMED 0;
   Key_1 : [CLASS_S] PACKED ARRAY[ l..U : INTEGER ] OF CHAR;
   Key_2 : [CLASS_S] PACKED ARRAY[ l2..u2 : INTEGER ] OF CHAR;
   Key_3 : [CLASS_S] PACKED ARRAY[ l3..u3 : INTEGER ] OF CHAR ): INTEGER;
                                                          EXTERNAL;
FUNCTION LBR$CLOSE( Library_Index : INTEGER ): INTEGER; EXTERNAL;

BEGIN  {Main program}
{
The call to LBR$INI_CONTROL initializes the library index and defines the
operation to be performed on the library.
}
Help_Stat := LBR$INI_CONTROL( Lib_Index, lbr$c_read,, );
IF NOT ODD( Help_Stat ) THEN LIB$STOP( Help_Stat );
{
The call to LBR$OPEN identifies the library to be accessed, in this case
the system help library, SYS$HELP:HELPLIB.HLB.
}
Help_Stat := LBR$OPEN( Library_Index := Lib_Index,
                       Fns := 'SYS$HELP:HELPLIB.HLB' );
IF NOT ODD( Help_Stat ) THEN LIB$STOP( Help_Stat );

{Get HELP keys:}
FOR Counter := 1 TO 3 DO
   BEGIN
   WRITE ('Enter key: ');
   {
The user is asked to supply the keys to look up help text.  In this case
Key[1] corresponds to a DCL command, while Key[2] and Key[3] (optional) are
command parameters or qualifiers.
   }
   READLN( Key[Counter] );
   END;  {FOR Counter}
{
If no routine name is specified in the call to LBR$GET_HELP, the
returned help text is written to SYS$OUTPUT (with line-width
defaulting to 80).  Note that several special symbols can be used for
the key arguments:

      *        All first-level help text in the library.

      KEY...   All help text with specified key and its subkeys.
```

(continued on next page)

Example 5–20 (Cont.): Accessing a System HELP Library

```
      *...      All help text in the library.
}
Help_Stat := LBR$GET_HELP( Lib_Index,,,, Key[1], Key[2], Key[3] );
   IF NOT ODD( Help_Stat ) THEN LIB$STOP( Help_Stat );
END.
```

The following is output when you run the program in Example 5–20:

```
$  RUN GET_HELP
Enter Key :  REQUEST
Enter Key :  /REPLY
Enter Key :  RETURN

  REQUEST

    /REPLY

     /REPLY

     Requests a reply to  the  specified  message,  and  issues
     a unique identification number to which the operator sends
     the response.
```

Example 5–21 shows the use of SYS$ASCTIM and SYS$GETTIM.

Example 5–21: Using SYS$ASCTIM and SYS$GETTIM

```
{
Source File: SYS$ASCTIM_AND_GETTIM.PAS
This program shows how to use $GETTIM and $ASCTIM.
}
[INHERIT( 'sys$library:starlet' )] PROGRAM Get_Time( OUTPUT );
VAR
   Date_String : VARYING[132] OF CHAR;
   Date_Time : RECORD
      l1, l2 : INTEGER;
      END;
BEGIN
$GETTIM( Date_Time );
$ASCTIM( Date_String.LENGTH, Date_String.BODY, Date_Time );
WRITELN( Date_String );
END.
```

Example 5–22 shows the use of SYS$CHECK_ACCESS.

Example 5–22: Using SYS$CHECK_ACCESS

```
{
Source File: SYS$CHECK_ACCESS.PAS
This program shows how to use the SYS$CHECK_ACCESS routine.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Use_Check_Access ( INPUT, OUTPUT );
TYPE
   Item_List_Cell = RECORD
      CASE INTEGER OF
         1 :    ( Buflen     : [WORD] 0..65535;
                  Code       : [WORD] 0..65535;
                  Bufadr     : INTEGER;
                  Retlenadr  : INTEGER );
         2 :    ( Terminator : INTEGER );
         END;
   Item_List_Template( Count : INTEGER ) =
      ARRAY[1..Count] OF Item_List_Cell;
VAR
   Item_List     : Item_List_Template( 2 );
   File_Name     : VARYING[132] OF CHAR;
   Return_Status : INTEGER;
   {
Make the variable VOLATILE since it changes as a  side-effect of the
call to $CHECK_ACCESS.
   }
   Privs_Used : [VOLATILE] PRIVS_USED_BITS$TYPE;

BEGIN
WRITE( 'Input the name of the file: ' );
READLN( File_Name );

{The item list is not optional, but  all items are:}
Item_List[1].Buflen     := SIZE( Privs_Used );
Item_List[1].Code       := chp$_privused;
Item_List[1].Bufadr     := IADDRESS( Privs_Used );
Item_List[1].Retlenadr  := 0;

Item_List[2].Terminator := 0;

Return_Status := $CHECK_ACCESS(
                    objtyp := acl$c_file,
                    objnam := file_name,
                    usrnam := 'BLANCHE',  { Or whatever... }
                    itmlst := item_list );
```

(continued on next page)

Example 5–22 (Cont.): Using SYS$CHECK_ACCESS

```
IF Return_Status = ss$_nopriv THEN
   WRITELN( 'SYS$CHECK_ACCESS returned SS$_NOPRIV' )
ELSE IF Return_Status = SS$_NORMAL THEN
        BEGIN
        WRITELN( 'SYS$CHECK_ACCESS returned SS$_NORMAL' );
        WRITELN( 'Privileges used: ' );
        WRITELN( '    SYSPRV  : ', Privs_Used.chp$v_sysprv );
        WRITELN( '    GRPPRV  : ', Privs_used.chp$v_grpprv );
        WRITELN( '    BYPASS  : ', Privs_used.chp$v_bypass );
        WRITELN( '    READALL : ', Privs_used.chp$v_readall );
        END
     ELSE LIB$SIGNAL( Return_Status );
END.
```

Example 5–23 shows the use of SYS$DEVICE_SCAN.

Example 5–23: Using SYS$DEVICE_SCAN

```
{
Source File: SYS$DEVICE_SCAN.PAS
This program shows how to use the SYS$DEVICE_SCAN routine.}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Use_Sys$Device_Scan( INPUT, OUTPUT );
TYPE
   Item_List_Cell = RECORD
      CASE INTEGER OF
         1: ( Buffer_Length : [WORD] 0..65535;
              Item_Code     : [WORD] 0..65535;
              Buffer_Addr   : INTEGER;
              Ret_Addr      : INTEGER; );
         2: ( Terminator    : INTEGER; );
      END;
   Item_List_Template( Count : INTEGER ) =
      ARRAY[1..Count] OF Item_List_Cell;
VAR
   Item_List   : Item_List_Template( 2 );
   Device_Name : VARYING[32] OF CHAR;
   Devclass    : INTEGER;
   Status      : INTEGER;
   IOSB        : ARRAY[1..2] OF INTEGER;
   Context : [QUAD] RECORD
      l1, l2 : INTEGER
      END;
```

(continued on next page)

Example 5–23 (Cont.): Using SYS$DEVICE_SCAN

```
BEGIN
WRITELN( 'Find all disks on the system' );
Devclass := dc$_disk;
Item_List[1].Buffer_Length := 4;
Item_List[1].Item_Code     := dvs$_devclass;
Item_List[1].Buffer_Addr   := IADDRESS( Devclass );
Item_List[1].Ret_Addr      := 0;
Item_List[2].Terminator    := 0;
Context := ZERO;

REPEAT
   Status := $DEVICE_SCAN( Device_Name.BODY,
             Device_Name.LENGTH,
             '*',
             Item_List,
             Context );
   IF ODD( Status ) THEN WRITELN( 'Device name: ', Device_Name )
   ELSE IF Status <> ss$_nomoredev THEN LIB$STOP( Status );
UNTIL Status = ss$_nomoredev;
END.
```

Example 5–24 shows the use of SYS$FAO.

Example 5–24: Using SYS$FAO

```
{
Source File: SYS$FAO.PAS
This program demonstrates the use of $FAO and $FAOL from VAX Pascal.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM FAO_Demo( OUTPUT );
VAR
   i             : INTEGER;
   Return_Status : INTEGER;
   Ctrstr        : VARYING[80] OF CHAR;
   Outbuf        : VARYING[133] OF CHAR;
   Sentence      : VARYING[50] OF CHAR;
   Prmlst        : ARRAY[1..50] OF INTEGER;

BEGIN
{Format an integer:}
i := 2525;
Ctrstr := 'The value of i is !ZL';
```

(continued on next page)

Example 5–24 (Cont.): Using SYS$FAO

```
Return_Status := $FAO( Ctrstr := Ctrstr,
                       Outlen := Outbuf.LENGTH,
                       Outbuf := Outbuf.BODY,
                       p1     := %IMMED i );
IF NOT ODD( Return_Status ) THEN LIB$STOP( Return_Status );
WRITELN( Outbuf );
Ctrstr := 'The magic string is "!AS"';
Return_Status := $FAO( Ctrstr := Ctrstr,
                       Outlen := Outbuf.LENGTH,
                       Outbuf := Outbuf.BODY,
                       p1     := %STDESCR 'VAX Pascal' );
IF NOT ODD( Return_Status) THEN LIB$STOP( Return_Status );
WRITELN( Outbuf );
Sentence := 'Goddess, do we have extensions!';

Return_Status := $FAO( Ctrstr := Ctrstr,
                       Outlen := Outbuf.LENGTH,
                       Outbuf := Outbuf.BODY,
                       p1     := %STDESCR ( sentence ) );
IF NOT ODD( Return_Status ) THEN LIB$STOP( Return_status );
WRITELN( Outbuf );
{Format the data/time:}
Ctrstr := 'The current time and date is !%D';

Return_Status := $FAO( Ctrstr := Ctrstr,
                       Outlen := Outbuf.LENGTH,
                       Outbuf := Outbuf.BODY,
                       p1     := %IMMED 0 );

IF NOT ODD( Return_Status ) THEN LIB$STOP( Return_Status );
WRITELN( outbuf );
{Format with 2 integers:}
Ctrstr := 'I = !SL, (I-3) = !SL';

Return_Status := $FAO( {Ctrstr :=} Ctrstr,
                       {Outlen :=} Outbuf.LENGTH,
                       {Outbuf :=} Outbuf.BODY,
                       {p1     :=} %IMMED i,
                       {p2     :=} %IMMED (i-3) );
IF NOT ODD( Return_Status ) THEN LIB$STOP( Return_Status );
WRITELN( Outbuf );
{Now lets try some faol examples:}
Ctrstr := 'Time is !%D, I = !SL, (I-3) = !SL';

{Fill info into prmlst:}
Prmlst[1] := 0;    {System Date/Time}
Prmlst[2] := i;
Prmlst[3] := i - 3;
```

(continued on next page)

Example 5–24 (Cont.): Using SYS$FAO

```
Return_Status := $FAOL( Ctrstr := Ctrstr,
                        Outlen := Outbuf.LENGTH,
                        Outbuf := Outbuf.BODY,
                        Prmlst := Prmlst );
IF NOT ODD( Return_Status) THEN LIB$STOP( Return_Status );
WRITELN( Outbuf );
{ Print out a character string:}
Ctrstr := '!AD!%D';
Sentence := 'The date and time is ';

{Fill info into prmlst:}
Prmlst[1] := Sentence.LENGTH;           {The length of the string}
Prmlst[2] := IADDRESS( Sentence.BODY);  {The address of the varying
                                         string's body}
Prmlst[3] := 0;                         {Date/Time}

Return_Status := $FAOL( Ctrstr := Ctrstr,
                        Outlen := Outbuf.LENGTH,
                        Outbuf := Outbuf.BODY,
                        Prmlst := Prmlst );
IF NOT ODD( Return_Status ) THEN LIB$STOP( Return_Status );
WRITELN( Outbuf );
END.
```

Example 5–25 shows the use of SYS$GETDVI.

Example 5–25: Using SYS$GETDVI

```
{
Source File: SYS$GETDVI.PAS
This program demonstrates how to use item lists with system services.
You can substitute any item-list-oriented system service for GETDVIW.
}
[INHERIT( 'sys$library:starlet' )]
PROGRAM Use_Getdviw( OUTPUT );
TYPE
   Item_List_Cell = RECORD
      CASE INTEGER OF
         1 : (( Normal Cell )
                Buffer_Length : [WORD] 0..65535;
                Item_Code     : [WORD] 0..65535;
                Buffer_Addr   : UNSIGNED;
                Return_Addr   : UNSIGNED );
         2 : (( Terminator )
                Terminator    : UNSIGNED );
      END;
   Item_List_Template( Count : INTEGER ) =
      ARRAY[1..Count] OF Item_List_Cell;
VAR
   Item_List        : Item_List_Template( 3 );
   Allocation_Class : [VOLATILE] INTEGER;
   Host_Name        : [VOLATILE] VARYING[15] OF CHAR;

BEGIN
{Fill in the item list cells:}
Item_List[1].Buffer_Length := SIZE( Host_Name.BODY );
Item_List[1].Item_Code     := dvi$_host_name;
Item_List[1].Buffer_Addr   := IADDRESS( Host_Name.BODY );
Item_List[1].Return_Addr   := IADDRESS( Host_Name.LENGTH );

Item_List[2].Buffer_Length := SIZE( allocation_class );
Item_List[2].Item_Code     := dvi$_alloclass;
Item_List[2].Buffer_Addr   := IADDRESS( allocation_class );
Item_List[2].Return_Addr   := 0;

Item_List[3].Terminator    := 0;

$GETDVIW( Devnam := 'SYS$DISK', Itmlst := item_list );

WRITELN( 'Allocation class = ', allocation_class:1 );
WRITELN( 'Host name = ', host_name );
END.
```

Example 5–26 shows the use of SYS$GETJPIW.

Example 5–26: Using SYS$GETJPIW

```
{
Source File: SYS$GETJPIW.PAS
This program demonstrates how to use item lists with system services.
You can substitute any item-list-oriented system service for GETJPIW.
}
[INHERIT( 'sys$library:starlet' )] PROGRAM Use_Getjpiw( OUTPUT );
TYPE
   Item_List_Cell = RECORD
      CASE INTEGER OF
         1 : ( { Normal Cell }
                Buffer_Length : [WORD] 0..65535;
                Item_Code     : [WORD] 0..65535;
                Buffer_Addr   : UNSIGNED;
                Return_Addr   : UNSIGNED );
         2 : ( { Terminator }
                Terminator  : UNSIGNED );
      END;
   Item_List_Template( Count : INTEGER ) =
      ARRAY[1..Count] OF Item_List_Cell;
VAR
   Item_List     : Item_List_Template( 4 );
   Process_Name  : [VOLATILE] VARYING[15] OF CHAR;
   Buffered_IO   : [VOLATILE] INTEGER;
   Current_Privs : [VOLATILE, QUAD] RECORD
      l1, l2 : UNSIGNED;
      END;

BEGIN
{Fill in the item list cells:}
Item_List[1].Buffer_Length := SIZE( Process_Name.BODY );
Item_List[1].Item_Code     := jpi$_prcnam;
Item_List[1].Buffer_Addr   := IADDRESS( Process_Name.BODY );
Item_List[1].Return_Addr   := IADDRESS( Process_Name.LENGTH );

Item_List[2].Buffer_Length := SIZE( Buffered_IO );
Item_List[2].Item_Code     := jpi$_bufio;
Item_List[2].Buffer_Addr   := IADDRESS( Buffered_IO );
Item_List[2].Return_Addr   := 0;

Item_List[3].Buffer_Length := SIZE( Current_Privs );
Item_List[3].Item_Code     := jpi$_curpriv;
Item_List[3].Buffer_Addr   := IADDRESS( Current_Privs );
Item_List[3].Return_Addr   := 0;

Item_List[4].Terminator := 0;

$GETJPIW( Itmlst := Item_List );
```

(continued on next page)

Example 5–26 (Cont.): Using SYS$GETJPIW

```
WRITELN( 'Process Name = ', Process_Name );
WRITELN( 'Buffered IO  = ', Buffered_IO:1 );
WRITELN( 'Current Privs= ', BIN( Current_Privs ) );
END.
```

Example 5–27 shows the use of SYS$GETQUI.

Example 5–27: Using SYS$GETQUI

```
{
Source File: SYS$GETQUI.PAS
This program shows how to use the SYS$GETQUI routine.
}
[INHERIT( 'sys$library:starlet' )] PROGRAM Getqui( INPUT, OUTPUT );
TYPE
   Item_List_Cell = RECORD
      CASE INTEGER OF
         1 : ( Buffer_Length : [WORD] 0..65535;
               Item_Code     : [WORD] 0..65535;
               Buffer_Addr   : INTEGER;
               Ret_Len_Addr  : INTEGER );
         2 : ( Terminator    : INTEGER );
      END;
   Item_List_Template( Count : INTEGER ) =
      ARRAY[1..Count] OF Item_List_Cell;
VAR
   Queue_Name        : [VOLATILE] VARYING[32] OF CHAR;
   Result_Queue_Name : [VOLATILE] VARYING[32] OF CHAR;
   Result_Job_Name   : [VOLATILE] VARYING[32] OF CHAR;
   Ret_Status        : INTEGER;
   Itmlst            : Item_List_Template( 3 );
   Job_Itmlst        : Item_List_Template( 3 );
   My_Search_Flags   : [VOLATILE] SEARCH_FLAGS$TYPE; {From STARLET}
   Job_Iosb, Iosb : RECORD
      Jbc_Status : INTEGER;
      Reserved   : INTEGER;
      END;

BEGIN
{Define the queue to search:}
Queue_Name := '*';
Itmlst[1].Buffer_Length := Queue_Name.LENGTH;
Itmlst[1].Item_Code     := qui$_search_name;
Itmlst[1].Buffer_Addr   := IADDRESS( Queue_Name.BODY );
Itmlst[1].Ret_Len_Addr  := 0;
```

(continued on next page)

Example 5–27 (Cont.): Using SYS$GETQUI

```
{Get the resultant name:}
Itmlst[2].Buffer_Length := SIZE( Result_Queue_Name.BODY );
Itmlst[2].Item_Code     := qui$_queue_name;
Itmlst[2].Buffer_Addr   := IADDRESS( Result_Queue_Name.BODY );
Itmlst[2].Ret_Len_Addr  := IADDRESS( Result_Queue_Name.LENGTH );

Itmlst[3].Terminator := 0;

Job_Itmlst[1].Buffer_Length := SIZE( Result_Job_Name.BODY );
Job_Itmlst[1].Item_Code     := qui$_job_name;
Job_Itmlst[1].Buffer_Addr   := IADDRESS( Result_Job_Name.BODY );
Job_Itmlst[1].Ret_Len_Addr  := IADDRESS( Result_Job_Name.LENGTH );

My_Search_Flags := ZERO;
My_Search_Flags.qui$v_search_all_jobs := TRUE;
Job_Itmlst[2].Buffer_Length := SIZE( My_Search_Flags );
Job_Itmlst[2].Item_Code     := qui$_search_flags;
Job_Itmlst[2].Buffer_Addr   := IADDRESS( My_Search_Flags );
Job_Itmlst[2].Ret_Len_Addr  := 0;

Job_Itmlst[3].Terminator    := 0;

REPEAT
   Ret_Status := $GETQUIW( Func   := qui$_display_queue,
                           Iosb   := Iosb,
                           Itmlst := Itmlst );
   IF Iosb.Jbc_Status <> jbc$_nomoreque THEN
      BEGIN
      WRITELN( Result_Queue_Name );
      REPEAT
         $GETQUIW( Func   := qui$_display_job,
                   Iosb   := Job_Iosb,
                   Itmlst := Job_Itmlst);
         IF ( Job_Iosb.Jbc_Status <> jbc$_nomorejob ) AND
            ( Job_Iosb.Jbc_Status <> jbc$_nosuchjob ) THEN
            WRITELN( '    ', Job_iosb.Jbc_Status, Result_Job_Name );
      UNTIL ( Job_Iosb.Jbc_Status = jbc$_nomorejob ) OR
            ( Job_Iosb.Jbc_Status = jbc$_nosuchjob );
      END; {IF Iosb.}
UNTIL Iosb.Jbc_Status = jbc$_nomoreque;
END.
```

Example 5–28 shows the use of SYS$GETSYI.

Example 5–28: Using SYS$GETSYI

```
{
Source File: SYS$GETSYI.PAS
This program shows how to use the SYS$GETSYI routine.
}
[INHERIT( 'sys$library:starlet' )] PROGRAM Use_Getsyi( OUTPUT );
TYPE
   Item_List_Cell = RECORD
      CASE INTEGER OF
         1 : ( Buffer_Length : [WORD] 0..65535;
               Item_Code     : [WORD] 0..65535;
               Buffer_Addr   : INTEGER;
               Len_Addr      : INTEGER );
         2 : ( Terminator    : INTEGER );
      END;
   Item_List_Template( Count : INTEGER ) =
      ARRAY[1..Count] OF Item_List_Cell;
VAR
   Context       : UNSIGNED;
   Nodename      : [VOLATILE] VARYING[32] OF CHAR;
   Itemlist      : Item_List_Template( 2 );
   Status_Return : INTEGER;

BEGIN
Itemlist[1].Buffer_Length := SIZE( Nodename.BODY );
Itemlist[1].Item_Code     := syi$_nodename;
Itemlist[1].Buffer_Addr   := IADDRESS( Nodename.BODY );
Itemlist[1].Len_Addr      := IADDRESS( Nodename.LENGTH );
Itemlist[2].Terminator    := 0;

Context := -1;
{Find all nodes in the VAXcluster:}
REPEAT
   Status_Return := $GETSYIW( Csidadr := Context,
                              Itmlst := Itemlist );
   IF Status_Return <> ss$_nomorenode THEN
      WRITELN( Nodename );
UNTIL Status_Return <> ss$_normal;
END.
```

Example 5–29 shows the use of SYS$GETUAI.

Example 5–29: Using SYS$GETUAI

```
{
Source File: SYS$GETUAI.PAS
This program shows how to use the SYS$GETUAI routine.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Use_Getuai( INPUT, OUTPUT );
TYPE
   Item_List_Cell = RECORD
      CASE INTEGER OF
         1 : ( Buffer_Length : [WORD] 0..65535;
               Item_Code     : [WORD] 0..65535;
               Buffer_Addr   : INTEGER;
               Ret_Len_Addr  : INTEGER );
         2 : ( Terminator    : INTEGER );
      END;
   Item_List_Template( Count : INTEGER ) =
      ARRAY[1..Count] OF Item_List_Cell;
VAR
   Username       : VARYING[12] OF CHAR;
   Item_List      : Item_List_Template( 2 );
   Stat           : INTEGER;
   UIC : RECORD
      member, group : [WORD] 0..65535;
      END;

BEGIN
Itemlist[1].Item_Code       := uai$_uic;
Itemlist[1].Buffer_Length   := SIZE( UIC );
Itemlist[1].Buffer_Addr     := IADDRESS( UIC );
Itemlist[1].Return_Len_Addr := 0;

Itemlist[2].Terminator      := 0;

WRITE( 'Which username?  ' );
READLN( Username );

Stat := $GETUAI( Usrnam := Username, Itmlst := Item_List );

IF NOT ODD( Stat ) THEN LIB$SIGNAL( Stat );

WRITELN( 'The UIC for ', Username, ' is [',
         OCT( UIC.Group, 3, 3 ), ',',
         OCT( UIC.Member, 3, 3 ), ']' );
END.
```

Example 5–30 shows the use of SYS$PROCESS_SCAN.

Example 5–30: Using SYS$PROCESS_SCAN

```
{
Source File: SYS$PROCESS_SCAN.PAS
This program shows how to use the SYS$PROCESS_SCAN routine.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Use_Sys$Process_Scan( INPUT, OUTPUT );
TYPE
   Item_List_Cell = RECORD
      CASE INTEGER OF
         1 : ( Buffer_Length : [WORD] 0..65535;
               Item_Code     : [WORD] 0..65535;
               Buffer_Addr   : INTEGER;     {Also Item_Value}
               Item_Flags    : INTEGER; );  {Also Ret_Addr}
         2 : ( Terminator    : INTEGER; );
      END;
   Item_List_Template( Count : INTEGER ) =
      ARRAY[1..Count] OF Item_List_Cell;
VAR
    Item_List    : Item_List_Template( 2 );
    Prefix       : [VOLATILE] VARYING[32] OF CHAR;
    Process_Name : [VOLATILE] VARYING[32] OF CHAR;
    Context      : UNSIGNED;
    Status       : INTEGER;
    Iosb         : ARRAY[1..2] OF INTEGER;

BEGIN
{
Find all the process on the system with process names starting
with DECW.
}
WRITELN( 'Find all processes whose process names begin with DECW' );
Prefix := 'DECW';
Item_List[1].Buffer_Length := LENGTH( Prefix );
Item_List[1].Item_Code     := pscan$_prcnam;
Item_List[1].Buffer_Addr   := IADDRESS( Prefix.BODY );
Item_List[1].Item_Flags    := pscan$m_prefix_match;
Item_List[2].Terminator    := 0;

Status := $PROCESS_SCAN( Context, Item_List );
IF NOT ODD( Status ) THEN LIB$STOP( Status );

{Set up itemlist for retrieving process information:}
Item_List[1].Buffer_Length := SIZE( Process_Name.BODY );
Item_List[1].Item_Code     := jpi$_prcnam;
Item_List[1].Buffer_Addr   := IADDRESS( Process_Name.BODY );
Item_List[1].Item_Flags    := IADDRESS( Process_Name.LENGTH );
Item_List[2].Terminator    := 0;
```

(continued on next page)

Example 5–30 (Cont.): Using SYS$PROCESS_SCAN

```
REPEAT
   Status := $GETJPIW( Itmlst := Item_list,
                       Pidadr := Context,
                       Iosb := iosb );
   IF ODD( Status ) THEN Status := Iosb[1];
   IF ODD( status ) THEN
      WRITELN( 'Process name: ', Process_Name )
   ELSE IF Status <> ss$_nomoreproc THEN LIB$STOP( Status );
UNTIL Status = ss$_nomoreproc;
END.
```

Example 5–31 shows the use of SYS$PUTMSG.

Example 5–31: Using SYS$PUTMSG

```
{
Source File: SYS$PUTMSG.PAS
This example shows how to use the SYS$PUTMSG routine.
}
[INHERIT( 'sys$library:starlet' )] PROGRAM Use_Putmsg( OUTPUT );
VAR
   Message_Vector : ARRAY[1..2] OF INTEGER;
{
Use CLASS_S attribute since that is the mechanism that  $PUTMSG
uses to hand the string to the routine.
}
FUNCTION Putmsg_Action_Routine(
   Message_Text :
   [CLASS_S] PACKED ARRAY[ 1..u : INTEGER] OF CHAR) : INTEGER;

   BEGIN
   WRITELN( 'Inside action routine with: ', Message_Text );
   {
   Return 1 to cause $PUTMSG to output the string.  Returning 0
   causes $PUTMSG not to output the string to SYS$OUTPUT.
   }
   Putmsg_Action_Routine := 1;
   END;

BEGIN
Message_Vector[1] := 1;    {Set argument count and default options }
Message_Vector[2] := ss$_abort;
$PUTMSG( Message_vector, %IMMED Putmsg_Action_Routine );
END.
```

Example 5–32 shows the use of SYS$SNDJBC.

Example 5–32: Using SYS$SNDJBC

```
{
Source File: SYS$SNDJBC.PAS
This program shows how to use the SYS$SNDJBC routine.
}
[INHERIT( 'sys$library:starlet', 'sys$library:pascal$lib_routines' )]
PROGRAM Use_$Sndjbcw( INPUT, OUTPUT );
TYPE
   Item_List_Cell = RECORD
      CASE INTEGER OF
         1 : ( Buffer_Length : [WORD] 0..65535;
               Item_Code     : [WORD] 0..65535;
               Buffer_Addr   : INTEGER;
               Return_Addr   : INTEGER; );
         2 : ( Terminator    : INTEGER; );
      END;
   Item_List_Template( Count : INTEGER ) =
      ARRAY[1..Count] OF Item_List_Cell;
VAR
   Item_List        : ARRAY[1..4] OF Itmlst_Cell;
   File_To_Print, Queue_To_Use : [VOLATILE] VARYING[132] OF CHAR;
   Stat             : INTEGER;
   Output_Status    : [VOLATILE] VARYING[255] OF CHAR;

BEGIN
WRITE( 'File to print? ' );
READLN( File_To_Print );
WRITE( 'Queue to use? ' );
READLN( Queue_To_Use );

{Enter single file into job:}
Itmlst[1].Buffer_Length := LENGTH( Queue_To_Use );
Itmlst[1].Item_Code     := sjc$_queue;
Itmlst[1].Buffer_Addr   := IADDRESS( Queue_To_Use.BODY );
Itmlst[1].Return_Addr   := 0;

Itmlst[2].Buffer_Length := LENGTH( File_To_Print );
Itmlst[2].Item_Code     := sjc$_file_specification;
Itmlst[2].Buffer_Addr   := IADDRESS( File_To_Print.BODY );
Itmlst[2].Return_Addr   := 0;

Itmlst[3].Buffer_Length := SIZE( Output_Status.BODY );
Itmlst[3].Item_Code     := sjc$_job_status_output;
Itmlst[3].Buffer_Addr   := IADDRESS( Output_Status.BODY );
Itmlst[3].Return_Addr   := IADDRESS( Output_Status.LENGTH );

Itmlst[4].Terminator    := 0;

Stat := $SNDJBCW( Func := sjc$_enter_file, Itmlst := Itmlst );
IF NOT ODD( Stat) THEN LIB$STOP( Stat );
```

(continued on next page)

Example 5–32 (Cont.): Using SYS$SNDJBC

```
WRITELN( Output_Status );

END.
```

5.8 DECwindows Example

Example 5–33 uses DECwindows routines and declarations.

Example 5–33: DECwindows Programming

```
{
Source file: DECWINDOWS.PAS
This example uses DECwindows routines and declarations to display a
dialog-box containing a push-button. Clicking on button once causes
the button's label to change; clicking again causes program to exit.
}
[INHERIT( 'sys$library:decw$dwtwidgetdef', {DECwindows Toolkit}
          'sys$library:starlet' )]         {System definitions}
PROGRAM Hello_World( OUTPUT );
TYPE
    Str = PACKED ARRAY[1..32] OF CHAR;
VAR
   Stat                 : UNSIGNED;
   Toplevel,
   HelloWorld_Main      : DWT$Widget;
   Argc                 : DWT$Cardinal VALUE 0;
   Arglist              : ARRAY[1..1] OF DWT$Arg;
   DRM_Hierarchy        : DWT$DRM_Hierarchy := NIL;
   Class                : DWT$DRM_Type;
   Regvec               : ARRAY[1..1] OF DWT$DRMreg_Arg;
   Regnum               : DWT$DRM_Count := 1;
   Hierarchy_Name_List  : ARRAY [1..1] OF ^DSC1$TYPE;
   Hierarchy_File_Name  : str := 'helloworld.uid';
   Button_Activate_Str  : str := 'helloworld_button_activate'(0);
```

(continued on next page)

Example 5–33 (Cont.): DECwindows Programming

```
PROCEDURE HelloWorld_Button_Activate( VAR Widget : DWT$Widget;
                                      VAR Tag    : INTEGER;
                                      VAR Reason : INTEGER );

   VAR
      Latin       : DWT$Comp_String;
      Call_Count  : [STATIC] INTEGER VALUE 0;
   BEGIN
   Call_Count := Call_Count + 1;
   IF Call_Count > 1 THEN
      $EXIT( 1 )
   ELSE
      BEGIN
      DWT$LATIN1_STRING( 'Goodbye'(13)'World'(13)'from Pascal!', Latin );
      DWT$VMS_SET_ARG( Latin, Arglist, 0, DWT$C_NLabel);
      XT$SET_VALUES( Widget, Arglist, 1 );
      XT$FREE( Latin )
      END;
   END; {helloworld_button_activate}

BEGIN
{Initialize the DRM:}
DWT$INITIALIZE_DRM;
{
Initialize the toolkit.  This call returns the id of the "toplevel"
widget.  The applications "main" widget must be the only child of
this widget.
}
Toplevel := XT$INITIALIZE( Name       := 'Hi',
                           Class_Name := 'helloworldclass',
                           URList     := 0,
                           Num_URList := 0,
                           ArgCount   := Argc );
{
Make sure the top-level widget allows resize so the button always
fits.
}
DWT$VMS_SET_ARG( Arg       := DWT$C_True,
                 ArgList   := Arglist,
                 ArgNumber := 0,
                 ArgName   := DWT$C_Nallow_Shell_Resize );
XT$SET_VALUES( Widget   := Toplevel,
               ArgList  := Arglist,
               ArgCount := 1 );
```

(continued on next page)

Example 5–33 (Cont.): DECwindows Programming

```
{Define the DRM hierarchy (only 1 file):}
NEW( Hierarchy_Name_List[1] );
WITH Hierarchy_Name_List[1]^ DO
   BEGIN
   DSC$B_Class      := DSC$K_Class_S;
   DSC$B_DType      := DSC$K_DType_T;
   DSC$W_MaxStrLen := LENGTH( Hierarchy_File_Name );
   DSC$A_Pointer    := IADDRESS( Hierarchy_File_Name )
   END;
Stat := DWT$Open_Hierarchy( Num_Files     := 1,
                            File_Names_List := Hierarchy_Name_List,
                            Hierarchy_Id_Return := DRM_Hierarchy,
                            Ancillary_Structures_List := %IMMED 0 );
IF Stat <> DWT$C_DRM_Success THEN
   BEGIN
   WRITELN( 'Can''t open hierarchy' );
   $EXIT;
   END;
{
Register our callback routines so that the resource manager can
resolve them at widget-creation time.
}
Regvec[1].DWT$A_DRMR_Name::INTEGER := IADDRESS( Button_Activate_Str );
Regvec[1].DWT$L_DRMR_Value := IADDRESS( HelloWorld_Button_Activate );
Stat := DWT$REGISTER_DRM_NAMES( Register_List := Regvec,
                                Register_Count := Regnum );
IF Stat <> DWT$C_DRM_Success THEN
   BEGIN
   WRITELN( 'Can''t register callback' );
   $EXIT
   END;

{Call DRM to fetch and create the pushbutton and its container:}
Stat := DWT$Fetch_Widget( Hierarchy_Id := DRM_Hierarchy,
                          Index        := 'helloworld_main',
                          Parent       := Toplevel,
                          W_Return     := HelloWorld_Main,
                          Class_Return := Class );
IF Stat <> DWT$C_DRM_Success THEN
   BEGIN
   WRITELN( 'Can''t fetch interface' );
   $EXIT
   END;
{
Make the toplevel widget "manage" the main window (or whatever the
uil defines as the topmost widget).  This causes it to be
"realized" when the toplevel widget is "realized."
}
XT$MANAGE_CHILD( HelloWorld_Main );
```

(continued on next page)

Example 5–33 (Cont.): DECwindows Programming

```
{
Realize the toplevel widget.  This causes the entire "managed" widget
hierarchy to be displayed.
}
XT$REALIZE_WIDGET( Toplevel );

{Loop and process events:}
XT$MAIN_LOOP
END. { helloworld }
```

VAX Pascal Glossary

actual discriminant

The boundary or selector value that you specify in a schema type to form a valid data type.

actual parameter

A value passed to a routine in the routine call.

alternate key

A key value in components of a file of indexed organization from which VAX Pascal provides an index into the file. Your program can use an alternate key to provide another collating sequence (order of access) for the file components.

argument

A name in the routine header that specifies information about the type, size, and passing mechanism of data that is expected to be passed to the routine as an actual parameter. This term is synonymous with the term *formal parameter*.

array

A group of components, called elements, that all have the same data type and share a common identifier.

attribute

An identifier that directs the VAX Pascal compiler to change its behavior in some way. Attributes are extensions to the Extended Pascal standard.

attribute class

A category that indicates a common affect that a group of attributes has on programming, such as data alignment, storage allocation, and optimization classes.

automatic variable allocation

An attribute of a variable that indicates that the variable be allocated each time the program enters the routine in which the variable is declared and is deallocated each time the program exits from that routine. A **set** is a collection of data items of the same ordinal type (called the **base type**)

base type

The data type of the data items in a set or of the object of a pointer. See also *pointer* and *set*.

cascade

An environment-file inheritance path that involves a compilation unit inheriting another compilation unit that inherits another compilation unit (and so forth). See also *compilation unit* and *environment file*.

case selector

An ordinal expression whose value at run time determines which statement in a CASE statement executes.

cells

A fixed-length file component in a file of relative file organization. Each cell is numbered consecutively from 1 to n.

compilation unit

A unit of code that can be compiled independently; the term compilation unit refers to either a program or a module.

compiler optimization

A process by which you produce source and object programs that achieve the greatest amount of processing with the least amount of time and memory.

component

A single data item in a file.

component access mode

A method by which VAX Pascal retrieves components from a file.

component format

A file characteristic that determines the size (or maximum size) of each component and any processing information needed in addition to the data portion of the component.

condition handler

A routine that is used to resolve an event, usually an error, that occurs during program execution and is detected by system hardware or software, or by the logic in a user application program.

constant expression

An expression that results in a value at the time you compile your program. See also *run-time expression*.

constructor

A list of values, surrounded by brackets ([]), used to assign values to structured objects, such as arrays, records, and sets.

control part

A data structure, internal to VAX Pascal, that contains information used by the compiler to create and to access the data part of nonstatic types at run time. See also *data part* and *nonstatic type*.

current component

The file component that is currently located in the file buffer variable; this is the only file component accessible to the program at a given time.

data part

A data structure that contains an object of a variable whose type is nonstatic. VAX Pascal usually accesses the data in the data part by accessing a pointer part that points to the object.

data type

A property of data that determines the range of values, set of valid operations, and maximum storage allocation for the data object.

declaration section

A part of a program, module, or routine that includes constant, label, type, variable, and routine declarations. A module can also contain an initialization (TO BEGIN DO) and a finalization (TO END DO) section. See also *executable section*, *heading* and *module*.

delayed device access

A VAX Pascal technique used to fill the file buffer. VAX Pascal reads and inserts a file component into the file buffer only when the program is ready to process it (when the program makes the next reference to the file).

direct access

A component access method that locates a file component according to either the random access number or the key value. See also *random access number* and *key*.

discriminated schema type

An expression that completes the boundary or range of a schema type, forming a valid data type.

element

A component of an array. See also *array*.

environment file

A file, created using the ENVIRONMENT attribute, that contains descriptions of the constant, type, variable, procedure, and function identifiers contained in the outermost level of a compilation unit. Compilation units that inherit this file, using the INHERIT attribute, have access to the data items declared in the other compilation unit. See also *attribute*, *compilation unit*, and *declaration section*.

executable section

A part of a program or routine containing statements to execute. See also *declaration section* and *heading*.

expression

A group of identifiers and operators that result in a value. See also *constant expression* and *run-time expression*.

extended-digit notation

A standard format for integers that includes the specification of a base value (bases 2 to 32 are allowed), followed by the number sign (#), and followed by the extended-digit value.

Extended Pascal standard

A proposed Pascal standard being prepared by a joint American (X3J9 /IEEE P770) and International (ISO IEC/JTC1/SC22/WG2) committee. VAX Pascal supports many features of this proposed standard, but not all of them.

extended-string format

A format for character-string constants that allows you to place nonprinting ASCII characters, such as the bell and the backspace, into the character string.

extensions

See *VAX Pascal extensions*.

external file

A physical file that has a name and exists outside the context of a VAX Pascal program. See also *file*.

field

A component of a record, which can be of various data types.

file

An organized collection of logically related data items.

file component

See *component*

file organization

A characteristic of a file that defines the physical arrangement of components within the file. See also *sequential file organization*, *relative file organization*, and *keyed file organization*.

fixed-length component format

A component format that specifies that all file components are the same length. See also *component* and *component format*.

formal discriminant

An indentifier in a schema-type declaration that take the place of specific boundary values or variant-record selectors. See also *schema type*.

formal parameter

A name in the routine header that specifies information about the type, size, and passing mechanism of data that is expected to be passed to the routine as an actual parameter. This term is synonymous with the term *argument*.

function

A subprogram that contains one or more statements to be executed once the function is called and that returns a single value.

generation mode

A file state that indicates when output is being written to a file. See also *inspection mode* and *undefined mode*.

heading

A part of a program, module, or routine that includes an identifier, a list of external files used (for programs and modules), a list of formal parameters (for routines), and a return value (for functions). See also *declaration section*, *executable section*, and *formal parameter*.

implementation module

A module that contains data to which you want to restrict access and that inherits an environment file from an interface module. See also *environment file* and *interface module*.

index

A internal data structure that provides pointers, based on key values, to file components in an indexed file.

indexed file organization

A file organization in which each file component must contain a primary key and, optionally, alternate keys. VAX Pascal uses the primary key to store components, and uses program-specified keys to build indexes and to retrieve data. See also *key*.

initial-state specifier

A constant expression, used with the reserved word VALUE, that initializes a variable, a constant, or a data type in the declaration section. See also *constant expression*.

inspection mode

A file state that indicates when input is being read from a file.

interface module

A module that produces an environment file, that contains data that is not likely to change, and that provides access to more restricted data in an implementation module. See also *environment file* and *implementation module*.

internal file

A file temporarily contained in memory that has no name and is not retained after the program finishes execution.

item list

A data structure that contains a sequence of control structures that provide input to a VMS system service and that describes where the service should place its output. An item list can have an arbitrary number of cells and is terminated with a longword of value 0.

key (or, key field)

A value in a component of a file of indexed organization that VAX Pascal uses to build indexes into the file. Each key is identified by its location within the component, its length, and its data type. See also *alternate key*, *index*, and *primary key*.

keyed access

Random file access by key value. See also *random access*.

key of reference

A key used by VAX Pascal to determine the index to use when sequentially accessing components of an indexed file. See also *key*, *indexed file organization*, and *sequential access method*.

label

A tag, declared in the LABEL declarations section, that makes an executable statement accessible to a GOTO statement.

language extensions

See *VAX Pascal extensions*.

lazy lookahead

See *delayed device access*

lexical elements

Characters and identifiers that have meaning to a compiler, such as the legal character set, special symbols, predeclared identifiers, and reserved words.

lock

Action taken by VAX Pascal that prevents other programs from accessing a file component while your program reads or writes that same component.

module

A set of instructions that can be compiled, but not executed, by itself. Module blocks contain only a declaration section, which can include an initialization (TO BEGIN DO) and a finalization (TO END DO) section.

module heading

See *module heading*.

multidimensional array

An array whose components are also arrays.

name string

A special form of constant expression required by some attributes. The name string is equivalent to a VAX Pascal character-string constant with one exception: name strings cannot use the extended-string syntax. See also *attribute* and *extended-string format*.

nonpositional syntax

A syntax for passing actual parameters that allows you to specify parameters in any order you want. The syntax requires that you specify the name of the formal parameter, followed by the assignment operator (:=), followed by the actual parameter.

nonstatic type

A type whose objects contain a run-time component; a type is nonstatic if it is a schema type or if its type is derived from a schema type.

optimization

See *compiler optimization*.

parameter-passing mechanism

The method by which VAX Pascal passes the actual parameter to the formal parameter. VAX Pascal passing mechanisms include passing by immediate value, by reference, and by descriptor.

parameter-passing semantics

The characteristics of a parameter expected by a routine declaration, as specified by the formal parameter. VAX Pascal passing semantics include value, variable, routine, and foreign parameters.

passing mechanism

See *parameter-passing mechanism*

positional syntax

A syntax for passing actual parameters that specifies that the parameters in the actual and formal lists must correspond exactly from left to right, item by item, through both lists.

predeclared identifier

A character string that is predeclared by the compiler to have a given meaning but that can be redefined in a program. VAX Pascal predeclared identifiers include names of data types, symbolic constants, file variables, procedures, and functions.

primary key

A key value in components of a file of indexed organization that indicates the order in which VAX Pascal stores the file components.

procedure

A subprogram that contains one or more statements to be executed once the procedure is called.

program

A set of instructions that can be compiled and executed by itself. Program blocks contain a declaration and an executable section.

program heading

See *heading*.

property

A characteristic of a program section (PSECT) that determines memory allocation and sharing. The term *property* is synonymous with the VMS term *attribute*.

random access

An access method that allows you to access a specified component in a relative or indexed file (and also in sequential files with fixed-length components). The order of access is not dependent on the order in which the components are stored.

record

A group of components, called fields, which can be of various data types.

recursion

The act of a routine directly or indirectly calling itself. See *recursion*.

redefinable reserved word

An identifier that VAX Pascal reserves for its own use but that you can redefine if you choose. If you redefine these words, the original function of the reserved word becomes unavailable within the block in which you redeclare the word.

relative component number

A cell number in a file of relative organization.

relative file organization

A file organization that consists of a series of component positions, called cells, numbered consecutively from 1 to n. The numbered, fixed-length cells enable VAX Pascal to calculate the component's physical position in the file.

reserved word

An identifier that VAX Pascal reserves for its use to designate data types, statements, and operators. You cannot redefine these identifiers.

routine

A subprogram; a function or procedure. See also *function* and *procedure*.

routine heading

See *heading*.

run-time expression

An expression that results in a value at the time you run your program. See also *constant expression*.

schema family

All types that are derived only from discriminating the same schema type, though the actual-discriminant values may vary. See also *actual discriminant*.

schema type

A user-defined construct that provides a template for a family of distinct data types. By discriminating a schema type, you create a valid data type. See also *data type*, *discriminated schema*, and *undiscriminated schema*.

semantics

See *parameter-passing semantics*.

sequential access method

A component access method in which storage or retrieval begins at a designated position in the file and continues through the file according to the component's position in storage.

sequential file organization

A file organization in which file components are stored one after the other, in the order in which they were written to the file.

set

A collection of data items of the same ordinal type. See also *base type*

short circuiting

Compiler evaluation of an expression from left to right that stops as soon as the overall result can be determined. See also *expression*.

static type

A type whose object can be fully described at compile time, a type that is not derived from a schema type.

static variable allocation

Allocation for a variable that occurs only once and that exists for the duration of the executable image's execution.

stream component format

A component format that is a continuous stream of bytes and that is delimited by a character called a terminator.

subscript

An ordinal index, or subscript, that designates an individual array element's position in the array. See also *array*.

terminator

A delimiting character for a stream component that VAX Pascal also recognizes as a valid part of the component data.

undefined mode

A file state that indicates when the file is in an undefined state of processing.

undiscriminated schema type

A schema type that has not been provided actual discriminants. These types are used as the domain type of a pointer or as a formal parameter.

unextended Pascal standards

Either of the following standards: American National Standard ANSI/IEEE770X3.97-1983 (ANSI) or International Standard ISO 7185-1983(E) (ISO). See also *Extended Pascal standard*.

user-action function

A function that you write and provide to VAX Pascal to use VMS Record Management Services (RMS) features to close a file.

variable-length component format

A component format that specifies that file components have lengths that vary. See also *component* and *component format*.

VAX Pascal extension

A lanuage element that is not part of either the unextended Pascal standards or the Extended Pascal standard. In the *VAX Pascal Reference Manual*, the term *extension*, refers to language elements that are not part of the Extended Pascal standard. See also *Extended Pascal standard* and *unextended Pascal standards*.

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[1]	————	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VAX Pascal User Manual
AA–H485F–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page Description

_____ _____

_____ _____

_____ _____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

Do Not Tear - Fold Here

Cut Along Dotted Line

Reader's Comments

VAX Pascal User Manual
AA–H485F–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

Do Not Tear - Fold Here

Cut Along Dotted Line

digital